# EXPLORE FLASK



by
Robert Picard

# 目錄

# Flask之旅

《explore flask》中文翻译

欢迎来到该译本的生产地点！任何bug report和别的pull request都会受到热烈欢迎！不要犹豫！

如果你想购买PDF格式的英文原本，前往http://exploreflask.com。

# 想参与？

如果你对本书有些建议，你可以在issue面板新开一个issue或提交一个pull request。如果你准备进行较大的改动，最好先在issue中提及，这样我们好能事先讨论这么做有没有必要。

译者：如果是关于翻译方面的问题，欢迎提issue或pull request。如果是关乎内容的问题，请移步https://github.com/rpicard/explore-flask

# 协议



# 感谢

由于本人水平有限，翻译过程中难免有错漏之处。感谢帮忙校正的各位网友：https://github.com/spacewander/explore-flask-zh/graphs/contributors

# 下载

见 https://github.com/spacewander/explore-flask-zh/wiki/%E4%B8%8B%E8%BD%BD

# 前言

本书旨在展示使用Flask的最佳实践。 开发一个普通的Flask应用需要跟形形色色的领域打交道。比如，你经常需要操作数据库，验证用户。 在接下来的几页里我将尽我所能来介绍处理这些事情时的“正确之道”。 我的建议并不总能有用，但我希望它们在大多数情况下都是一个好选择。

## 假设

为了给你提供更贴切的建议，我将基于几个基本的假设撰写本书。当你阅读并在自己的项目中应用这些建议时，请勿忘这一点。

## 受众

本书的内容基于官方文档之上。我强烈建议你深入阅读官方用户指南和新手教程。这将给你一个更熟悉Flask的机会。你至少需要知道什么是view，Jinja模板的基础知识以及新手应有的其他基本概念。我会尽量避免重提用户指南中存在的信息，所以如果直接阅读本书，你就会有对阅读官方文档的急迫需求（这不错吧？）。

虽然这么说，本书涉及的主题并不高深。本书仅仅是强调减轻开发者负担的最佳实践和模式。尽量避免啰嗦官方文档中提到的内容的同时，我也会再次强调一些概念来加深印象。 在阅读这部分内容时，你不需要重读新手教程。

# 版本

## Python 2 还是 Python 3

当我写下此文， Python社区正处于从Python 2迁移到Python 3的动荡之中。Python Software Foundation的官方态度如下：

> Python 2.x is the status quo, Python 3.x is the present and future of the language.Python wiki: python2还是python3

到了版本0.10，Flask现在可以在Python 3.3上运行。就新的Flask应用是否需要使用Python 3的问题，我问过Armin Ronacher，他回答说，这不是必须的：

> 我自己现在并不用它，我也不会向别人推荐自己都不相信的东西，所以我不会推荐Python 3. -- Armin Ronacher, Flask作者我和Armin Ronacher的对话

主要的理由在于许多常用的包没有Python 3的版本。 你总不会愿意接受用python 3开发了几个月后发现自己不能使用包X,Y,Z…… 也许总有一天Flask官方将推荐用Python 3开始新的项目，但是现在依然是Python 2的天下。

另注

Python 3 Wall of Superpowers记录了一些已经移植到Python 3的包。

既然本书需要提供实践上的建议，我将假定你正使用Python 2。 更准确地说，我将基于Python 2.7撰写本书。随着Flask社区的变迁，将来的更新会改变这一点，但是在当下，我们依然活在Python 2.7的世界里。

### Flask 版本 0.10

正当本书撰写之时，0.10是Flask的最新版本（准确说，是0.10.1版）。本书中大多数内容不会受到Flask的较小的变动的影响，但是你需要了解这一点。

## 持续集成

本书的内容将持续更新，而不是周期性发布。 这样做有一个好处，内容可以得到及时地更新，而不会待字闺中。 所以这个网站内容将会比印刷版甚至PDF更加前沿。

# 本书用到的约定

## 各章独立成文

本书的每一章独立成文。许多书和教程通篇浑然一体。通常这意味着，一个示范程序或一个应用的创建和更新将贯穿全书来展示概念和主题。 本书的范例将散布于每一章节来展示概念，但不意味着这些范例可以组合成一个大的项目。

## 格式

示例代码将以代码块形式来呈现。

```
print “Hello world!”
```

目录列表有时会被用来展示一个应用或目录的大致结构。

```
static/
  style.css
  logo.png
  vendor/
    jquery.min.js
```

脚注会用于引用中，这样就不会跟正文混乱起来了。

斜体将用来表示文件名。

粗体将用来表示新的或重要的内容。

> 注意 这里会有容易掉进去（而且会造成大问题）的坑。

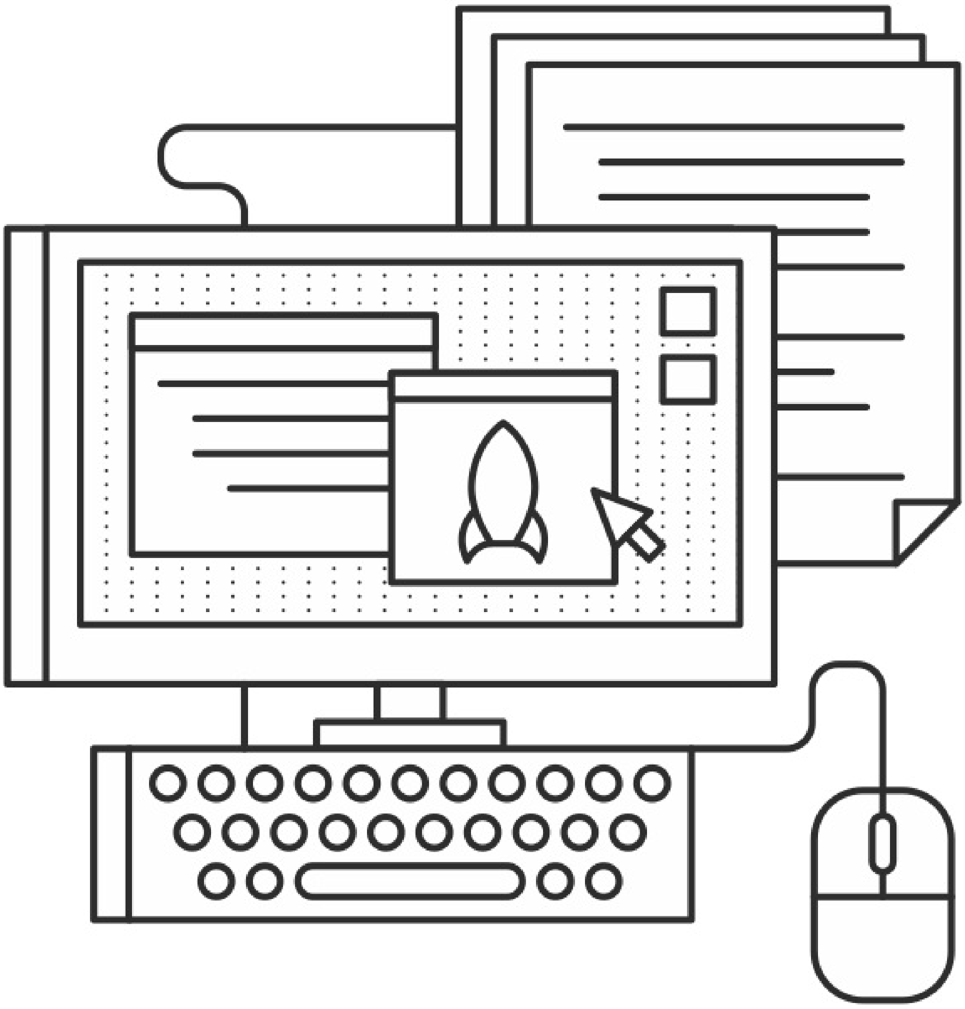
---

> 参见 这里会有一些补充信息。

# 总结

- 本书包含了使用Flask的最佳实践。
- 我假定你通读了Flask教程

- 本书基于Python 2.7
- 本书基于Flask 0.10
- 通过年度审查，我尽量让本书的内容保持更新。
- 本书中每一章独立成文。
- 我通过一些约定来表达跟内容相关的附加信息。
- 每章的结尾都会出现对本章内容的总结。



# 代码约定

在Python社区中有许多关于代码风格的约定。如果你写过一段时间Python了，那么也许对此已经有些了解。我会简单介绍一下，同时给你一些

URL链接，从中你可以找到关于这个话题的详细信息。

# 让我们提出一个PEP！

**PEP**全称是“Python Enhancement Proposal”（Python增强提案）。你可以在python.org上找到它们以及对应的索引目录。PEP在索引目录中按照数字编号排列，包括了元PEP（meta-PEP，讨论关于PEP的细节）。与之对应的是技术PEP（technical PEP），思考的是诸如Python内部实现的改良这样的话题。

有一些PEP，比如PEP 8和PEP 257，影响了Python代码风格的标准。 PEP 8包括了Python代码风格的规约。 而PEP 257包括了文档字符串（docstrings， 在Python中给代码加文档的标准方式）的规约。

# PEP 8: Python代码风格规约

PEP 8是对Python代码风格的官方规约。 我建议你阅读它并将之付诸在Flask项目（以及其他Python项目）的开发实践中。 当项目规模膨胀到多个包含成百上千行代码的文件时，这样做会使你的代码更加工整、了然。毕竟PEP 8的建议都是围绕着实现更加可读的代码这个目标。 另外，如果你的项目准备开源，潜在的奉献者（contributors）会很高兴看到你的代码是遵循PEP 8的。

一个至关重要的建议是每级缩进使用4个空格。不要使用tab。 如果你打破了这个规约，它将会成为你（以及你的队友）在项目间切换的一个负担。 这种不一致一向是任意语言心中的痛，但是对于Python，一门着重留白的语言，这是一个不可承受之重。 因为tab与space之间的混搭会导致不可预期且难以排查的错误。

## PEP 257: 文档字符串规约

PEP 257 覆盖了Python的另一项标准:**docstrings**。你可以阅读PEP中的定义和相关建议，不过这里会给一个例子来展示一个文档字符串应该是怎样的：

```
def launch_rocket():
    """主要的火箭发射调度器

    启动发射火箭所需的每一个步骤。
    """
    # [...]
```

这种风格的文档字符串可以通过一些诸如Sphinx的软件来生成不同格式的文档。同时它们也有助于让你的代码更加工整。

> 参见
>
> - PEP 8 http://legacy.python.org/dev/peps/pep-0008/
> - PEP 257 http://legacy.python.org/dev/peps/pep-0257/
> - Sphinx http://sphinx-doc.org/，一个文档生成器，同出于Flask作者之手

# 相对形式的import

开发Flask应用时，使用相对形式的import会让你的生活更加轻松。 原因很简单。之前，当需要import一个内部模块时，你也许要显式指明应用的包名（the app's package name）。假设你想要从*myapp/models.py*中导入 `User` 模型：

```
# 使用绝对路径来导入User
from myapp.models import User
```

用了相对形式的import后，你可以使用点标记法：第一个 `.` 来表示当前目录，之后的每一个 `.` 表示下一个父目录。

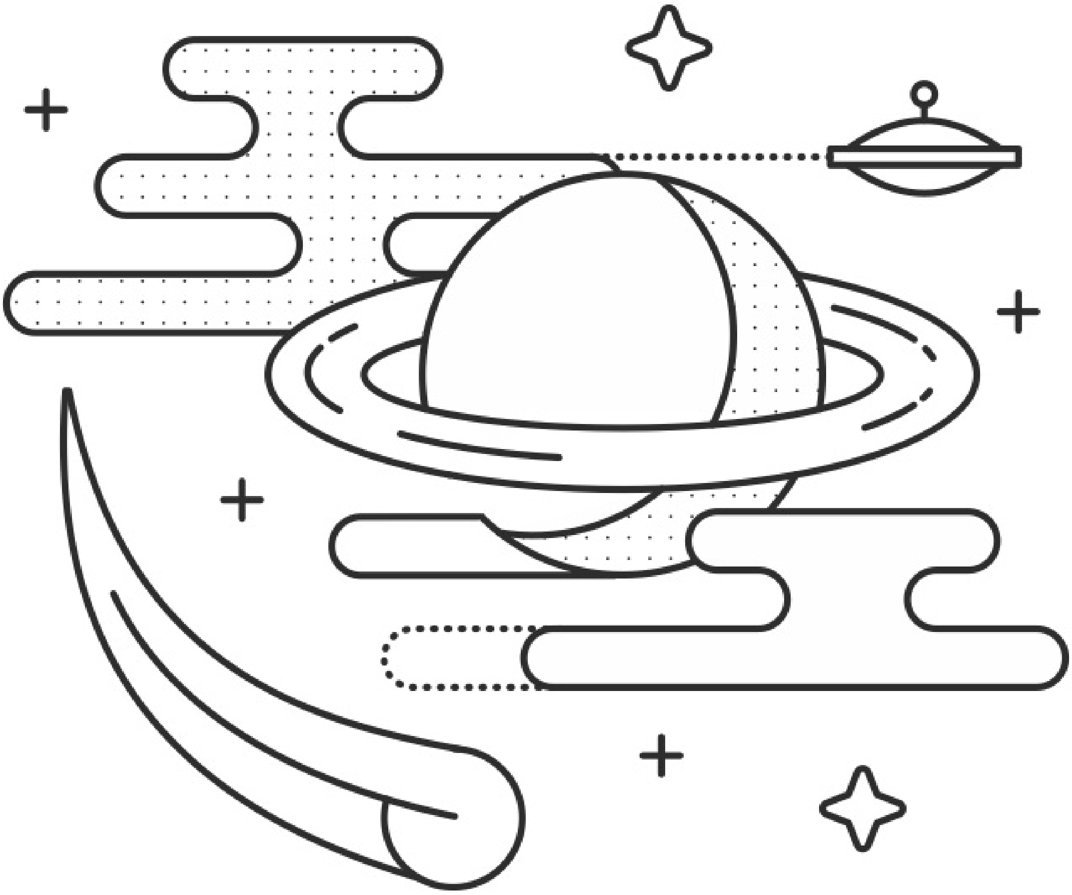
```
# 使用相对路径来导入User
from .models import User
```

这种做法的好处在于使得package变得更加模块化了。现在你可以重命名你的package并在别的项目中重用模块，而无需忍受更新被硬编码的包名之苦。

> 参见
>
> - 你可以在PEP 328的这一节里读到更多关于相对形式的import的语法
> - 在写作本书的过程中，我碰巧在这个Tweet上面看到了一个使用相对形式的import的好处：https://twitter.com/dabeaz/status/372059407711887360 Just had to rename our whole package. Took 1 second. Package relative imports FTW!

# 总结

- 尽量遵循PEP 8中的代码风格规约。
- 尽量遵循PEP 257中的文档字符串规约。
- 使用相对形式的import来import你的应用中的内部模块。



# 环境

为了正确地跑起来，你的应用需要依赖许多不同的软件。就算是再怎么否认这一点的人，也无法否认至少需要依赖Flask本身。你的应用的运行环

境，在当你想要让它跑起来时，是至关重要的。幸运的是，我们有许多工具可以减低管理环境的复杂度。

# 使用virtualenv来管理环境

virtualenv是一个能把你的应用隔离在一个虚拟环境中的工具。一个虚拟环境是一个包含了你的应用依赖的软件的文件夹。一个虚拟环境同时也封存了你在开发时的环境变量。与其把依赖包，比如Flask，下载到你的系统包管理文件夹，或用户包管理文件夹，我们可以把它下载到对应当前应用的一个隔离的文件夹之下。这使得你可以指定一个特定的Python二进制版本，取决于当前开发的项目。

virtualenv也可以让你给不同的项目指定同样的依赖包的不同版本。当你在一个老旧的包含众多不同项目的平台上开发时，这种灵活性十分重要。

用了virtualenv，你将只会把少数几个Python模块安装到系统的全局空间中。其中一个会是virtualenv本身：

```
# 使用pip安装virtualenv
$ pip install virtualenv
```

安装完virtualenv，就可以开始创建虚拟环境。切换到你的项目文件夹，运行 `virtualenv` 命令。这个命令接受一个参数，作为虚拟环境的名字（同样也是它的位置，在当前文件夹 `ls` 下你就知道了）。

```
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip...done.
```

这将创建一个包含所有依赖的文件夹。

一旦新的virtual environment已经准备就绪，你需要给对应的virtual environment下的 `bin/activate` 脚本执行 `source` ，来激活它。

```
$ source venv/bin/activate
```

你可以通过运行 `which python` 看到："python"现在指向的是virtual environment中的二进制版本。

```
$ which python
/usr/local/bin/python
$ source venv/bin/activate
(venv)$ which python
/Users/robert/Code/myapp/venv/bin/python
```

当一个virtual environment被激活了，依赖包会被pip安装到virtual environment中而不是全局系统环境。

你也许注意到了，你的shell提示符发生了改变。virtualenv在它前面添加了当前被激活的virtual environment，所以你能意识到你并不存在于全局系统环境中。

运行 `deactivate` 命令，你就能离开你的virtual environment。

```
(venv)$ deactivate
$
```

# 使用virtualenvwrapper管理你的virtual environment

我想要让你了解到virtualenvwrapper对于前面的工作做了什么改进，这样你就知道为什么你应该使用它。

虚拟环境文件夹现在已经位于你的项目文件夹之下。 但是你仅仅是在激活了虚拟环境后才会跟它交互。它甚至不应该被放入版本控制中。所以它呆在项目文件夹中也是挺碍眼的。 解决这个问题的一个方法就是使用virtualenvwrapper。这个包把你所有的virtual environment整理到单独的文件夹下，通常是 `~/.virtualenvs/` 。

要安装virtualenvwrapper，遵循这个文档中的步骤。

> 注意 确保你在安装virtualenvwrapper时不在任何一个virtual environment中。你需要把它安装在全局空间，而不是一个已存在的virtual environment中

现在，你不再需要运行 `virtualenv` 来创建一个环境，只需运行 `mkvirtualenv` ：

```
$ mkvirtualenv rocket
New python executable in rocket/bin/python
Installing setuptools, pip...done.
```

在你的virtual environment目录之下，比如在 `~/.virtualenv` 之下， `mkvirtualenv` 创建了一个文件夹，并替你激活了它。 就像普通的 `virtualenv` ， `python` 和 `pip` 现在指向的是virtual environment而不是全局系统。 为了激活想要的环境，运行这个命令： `workon [environment name]` ，而 `deactivate` 依然会关闭环境。

# 记录依赖变动

随着项目增长，你会发现它的依赖列表也一并随着增长。在你能运行一个Flask应用之前，即使已经需要数以十记的依赖包也毫不奇怪。管理依赖的最简单的方法就是使用一个简单的文本文件。pip可以生成一个文本文件，列出所有已经安装的包。它也可以解析这个文件，并在新的系统（或者新的环境）下安装每一个包。

## pip freeze

**requirements.txt**是一个常常被许多Flask应用用于列出它所依赖的包的文本文件。它是通过`pip freeze > requirements.txt`生成的。使用`pip install -r requirements.txt`，你就能安装所有的包。

> 注意 在freeze或install依赖包时，确保你正为于正确的virtual environment之中。

## 手动记录依赖变动

随着项目增长，你可能会发现`pip freeze`中列出的每一个包并不再是运行应用所必须的了。 也许有些包只是在开发时用得上。`pip freeze`没有判断力；它只是列出了当前安装的所有的包。所以你只能手动记录依赖的变动了。 你可以把运行应用所需的包和开发应用所需的包分别放入对应的*require_run.txt*和*require_dev.txt*。

# 版本控制

选择一个版本控制系统并使用它。 我推荐Git。如我所知，Git是当下最大众的版本控制系统。 在删除代码的时候无需担忧潜在的巨大灾难是无价的。 你现在可以对过去那种把不要的代码注释掉的行为说拜拜了，因为你可以直接删掉它们，即使将来突然需要，也可以通过`git revert`来恢复。 另外，你将会有整个项目的备份，存在Github,Bitbucket或你自己的Git server。

## 什么不应该在版本控制里

我通常不把一个文件放在版本控制里，如果它满足以下两个原因中的一个。

1. 它是不必要的
2. 它是不公开的。

编译的产物，比如 `.pyc` ，和virtual environment（如果你因为某些原因没有使用virtualenvwrapper）正是前者的例子。 它们不需要加入到版本控制中，因为它们可以通过 `.py` 或 `requirements.txt` 生成出来。

接口密钥（调用接口时必须传入的参数），应用密钥，以及数据库证书则是后者的例子。 它们不应该在版本控制中，因为一旦泄密，将造成严重的安全隐患。

> 注意 在做安全相关的决定时，我会假设稳定版本库将会在一定程度上被公开。这意味着要清除所有的隐私，并且永不假设一个安全漏洞不会被发现， 因为“谁能想到他们会干出什么事情？”

在使用Git时，你可以在版本库中创建一个特别的文件名为*.gitignore*。在里面，能使用正则表达式来列出对应的文件。任何匹配的文件将被Git所忽略。我建议你至少在其中加入 `*.pyc` 和 `/instance` 。instance文件夹中存放着跟你的应用相关的不便公开的配置。

```
.gitignore:
*.pyc
instance/
```

> 参见
>
> - 在这里你可以了解什么是*.gitignore*：http://git-scm.com/docs/gitignore
> - Flask文档中对instance目录的一段介绍：http://flask.pocoo.org/docs/config/#instance-folders

# 调试

## 调试模式

Flask有一个便利的特性叫做“debug mode”。在你的应用配置中设置 `debug = True` 就能启动它。 当它被启动后，服务器会在代码变动之后重新加载，并且一旦发生错误，错误会打印成一个带交互式命令行的调用栈。

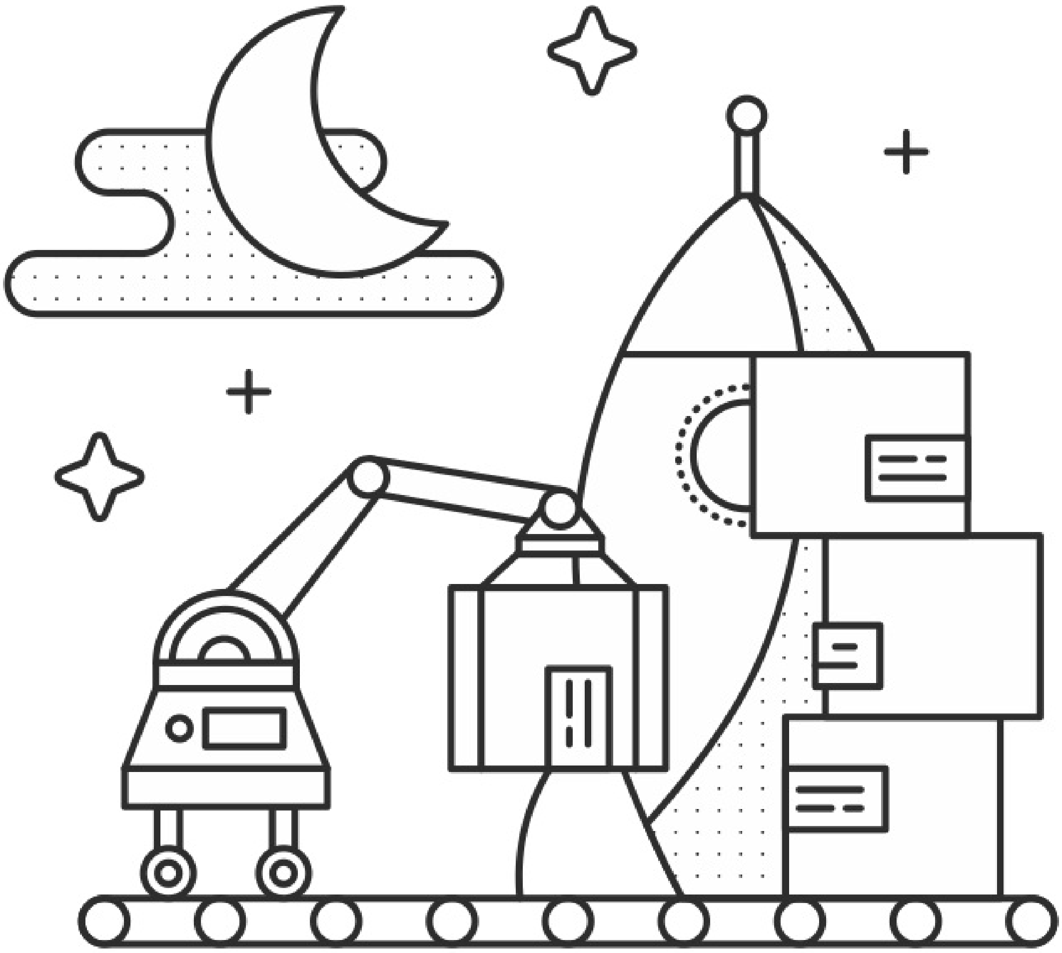
> 注意 不要在生产环境中开启debug mode。交互式命令行运行任意的代码输入，如果是在运行中的网站上，这将导致安全上的灾难性后果。
>
> 另见
>
> - 阅读一下quickstart页面的debug mode部分：http://docs.jinkan.org/docs/flask/quickstart.html#debug-mode
> - 这里有一些关于错误处理，日志记录和使用其他调试工具的信息：http://docs.jinkan.org/docs/flask/errorhandling.html

## Flask-DebugToolbar

Flask-DebugToolbar 是用来调试你的应用的另一个得力工具。在debug mode中，它在你的应用中添加了一个侧边条。 这个侧边条会给你提供有关SQL查询，日志，版本，模板，配置和其他有趣的信息。

## 总结

- 使用virtualenv来打包你的应用的依赖包。
- 使用virtualenvwrapper来打包你的virtual environment。
- 使用一个或多个文本文件来记录依赖变化。
- 使用一个版本控制系统。我推荐Git。
- 使用.gitignore来排除不必要的或不能公开的东西混进版本控制。
- debug mode会在开发时给你有关bug的信息。
- Flaks-DebugToolbar拓展将给你有关bug更多的信息。



# 组织你的项目

Flask会把项目组织的职责托付给你。 这是我喜欢使用Flask开始项目的其中一个理由，但是这意味着你不得不思考怎么组织你的代码。 你可以把这

个应用放到一个文件中，或者把它分割多个包。然而这两种结构并不适合大多数项目。 这里有一些固定的组织模式，你可以遵循它们以便于开发和部署。

# 约定

在这一段中我想要先约定一些概念。

版本库（**Repository**）：你的应用的根目录。这个概念来自于版本控制系统，但在这里有所拓展。 当我在这一章提到“版本库”时，指的是你的项目的根目录。在开发你的应用时，你不太可能会离开这个目录。

包（**Package**）：包含了你的应用代码的一个包。在这一章，我将深入探讨以包的形式建立你的应用，但是现在只需知道包是版本库的一个子目录。

模块（**Module**）：一个模块是一个简单的，可以被其它Python文件引入的Python文件。一个包由多个模块组成。

> 参见
>
> - 在这里可以读到更多的关于Python模块的内容: http://docs.python.org/2/tutorial/modules.html
> - 这个链接中也有一节关于包的内容: http://docs.python.org/2/tutorial/modules.html#packages

# 组织模式

## 单一模块

在许多Flask例子里，你会看到它们把所有的代码放到一个单一文件中，通常是*app.py*。对于一些微（~~写完就丢~~）项目来说这恰到好处，毕竟你只

需要处理几个路由（route）并且只有百来行代码。（示例用的应用就是这样）

单一模块的应用的版本库看起来像这样：

```
app.py
config.py
requirements.txt
static/
templates/
```

在这个例子中，应用逻辑部分会存放在*app.py*

# 包

当你开始在一个变得更加复杂的项目上工作时，单一模块就会造成严重的问题。你需要为模型（model）和表单（form）定义多个类，而它们会跟你的路由和配置代码又吵又闹。所有的一切让你焦头烂额。为了解决这个问题，我们得把应用中不同的组件分开到单独的、高内聚的一组模块 - 也即是包 - 之中。

基于包的应用的版本库看起来就像是这样：

```
config.py
requirements.txt
run.py
instance/
  /config.py
yourapp/
  /__init__.py
  /views.py
  /models.py
  /forms.py
  /static/
  /templates/
```

这个结构允许你理智地整理你的应用的不同组件。有关模型的类定义全待在*models.py*，而路由定义在*views.py*，有关表单的类定义全待在*forms.py*（我们等会会用整整一章的篇幅谈谈表单）。

下面的表格列举了大多数Flask应用都有的基本组件。对于你的应用，可能还需要别的一些文件，但这些适用于大多数Flask应用。

| 组件 | 作用 |
| --- | --- |
| | 这个文件中用于启动一个开发服务器。它从你的包获得应用的副本并运行 |

| | |
|---|---|
| run.py | 它。这不会在生产环境中用到，不过依然在许多Flask开发的过程中看到。 |
| requirements.txt | 这个文件列出了你的应用依赖的所有Python包。你可能需要把它分成生产依赖和开发依赖。[请看第三章] |
| config.py | 这个文件包含了你的应用需要的大多数配置变量 |
| instance/config.py | 这个文件包含不应该出现在版本控制的配置变量。其中有类似调用密钥和数据库URI连接密码。同样也包括了你的应用中特有的不能放到阳光下的东西。比如，你可能在*config.py*中设定 `DEBUG = False` ，但在你自己的开发机上的*instance/config.py*设置 `DEBUG = True` 。因为这个文件可以在*config.py*之后被载入，它将覆盖掉 `DEBUG = False` ，并设置 `DEBUG = True` 。 |
| | |

| | |
|---|---|
| yourapp/ | 这个包里包括了你的应用。 |
| yourapp/__init__.py | 这个文件初始化了你的应用并把所有其它的组件组合在一起。 |
| yourapp/views.py | 这里定义了路由。它也许需要作为一个包（*yourapp/views/*），由一些包含了紧密相联的路由的模块组成。 |
| yourapp/models.py | 在这里定义了应用的模型。你可能需要像对待*views.py*一样把它分割成许多模块。 |
| yourapp/static/ | 这个文件包括了公共CSS， Javascript, images和其他你想通过你的应用展示出去的静态文件。默认情况下人们可以从*yourapp.com/static/*获取这些文件。 |
| yourapp/templates/ | 这里放置着你的应用的Jinja2模板。 |

# Blueprints

有朝一日你可能会发觉应用里有许多相关的路由了。如果是我，我会首先把*views.py*分割成一个包并把相关的路由组织成模块。 要是你已经这么做了，是时候把你的应用分解成蓝图（blueprints）了

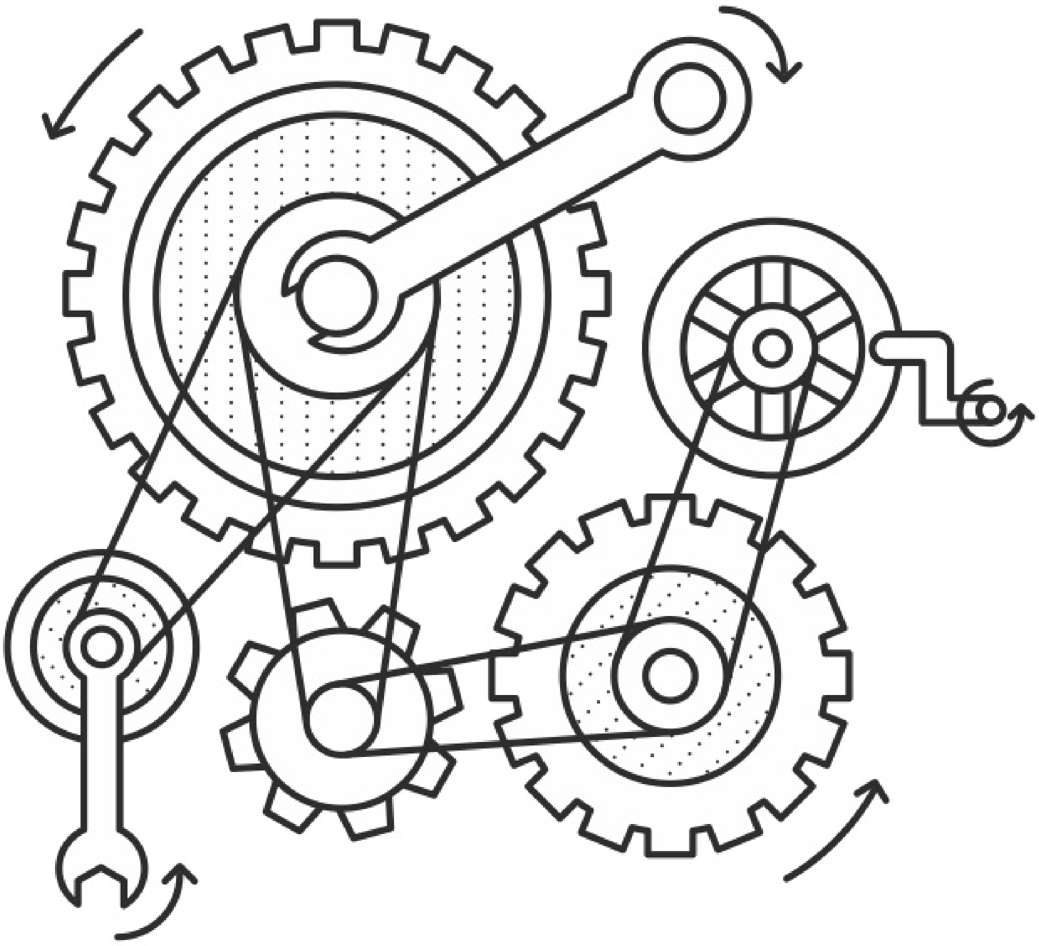
蓝图是按照一定程度上的自组织的方式，作为你的应用的一部分的组件。 它们表现得就像你的应用下的子应用一样。你可能使用不同的蓝图来对应管理面板（admin panel），前端（front-end）和用户面板（user dashboard）。 这使得你按照组件组织视图，静态文件和模板，并在组件间共享模型，表单和你的应用的其他部分。

你可以在第7章阅读到关于蓝图的更多内容。

# 总结

- 对于微应用，建议使用单一模块结构。
- 对于包含了视图，模型，表单以及更多的项目，使用包结构。

- 蓝图是把项目按照一些不同的组件组织起来的好办法。



# 配置

当你开始学习Flask时，配置看上去是小菜一碟。你仅仅需要在*config.py*定义几个变量，然后万事大吉。 然而当你不得不管理一个生产上的应用的

配置时，这一切将变得棘手万分。你不得不设法保护API密钥，或者纠结于为了不同的环境（比如开发环境和生产环境）使用不同的配置。在本章我们将探讨Flask的一些高级特性，它们能让配置管理更为轻松。

# 从小处起步

一个简单的应用不需要任何复杂的配置。你仅仅需要在你的根目录下放置一个*config.py*文件，并在*app.py*或*yourapp/__init__.py*中加载它。

*config.py*的每一行中应该是某一个变量的赋值语句。一旦*config.py*在稍后被加载，这个配置变量可以通过`app.config`字典来获取，比如`app.config["DEBUG"]`。以下是一个小项目的*config.py*文件的范例：

```
DEBUG = True # 启动Flask的Debug模式
BCRYPT_LEVEL = 13 # 配置Flask-Bcrypt拓展
MAIL_FROM_EMAIL = "robert@example.com" # 设置由
```

有一些配置变量是内建的，比如 `DEBUG` 。还有些配置变量是关于Flask拓展的，比如 `BCPYRT_LEVEL` 就是用于Flask-Bcrypt拓展（一个用于hash映射密码的拓展）。你甚至可以定义在这个应用中用到的自己的配置变量。在这个例子中，我使用 `app.config["MAIL_FROM_EMAIL"]` 来表示邮件往来时（比如重置密码）默认的发送方。这使得在将来要修改的时候不会带来太多麻烦。

为了加载这些配置变量，我通常使用 `app.config.from_object()` 。如果是单一模块应用中，是在*app.py*；或者在*yourapp/__init__.py*，如果是基于包的应用。无论在哪种情况下，代码看上去像这样：

```
from flask import Flask

app = Flask(__name__)
app.config.from_object('config')
# 现在通过app.config["VAR_NAME"]，我们可以访问到对
```

# 一些重要的配置变量

| 变量 | 描述 | 默认值 |
| --- | --- | --- |
| DEBUG | 在调试错误的时候给你一些有用的工具。比如当一个请求导致异常的发生时，会出现的一个web界面的调用堆栈和Python命令行。 | 在开发环境下应该设置成True，在生产环境下应设置为False。 |
| SECRET_KEY | Flask使用这个密钥来对cookies和别的东西进行签名。你应该在*instance*文件夹中设定这个值，并不要把它放入版本控制中。你可以在下一节 | 这应该是一个复杂的任意值。 |

| | | |
|---|---|---|
| | 读到关于*instance*文件夹的更多信息。 | |
| BCRYPT_LEVEL | 如果使用Flask-Bcrypt来hash映射用户密码（如果没有，现在就用它），你需要为hash密码的算法指定“rounds”的值。设置的rounds值越高，计算一次hash花费的时间就越长（同样的效果作用于破解方，这个才是重要的）。rounds的值应该随着你的设备的计 | 如果使用Flask-Bcrypt来hash映射用户密码（如果没有，现在就用它），你需要为hash密码的算法指定“rounds”的值。设置的rounds值越高，计算一次hash花费的时间就越长（同样的效果作用于破解方，这个才是重要的）。rounds的值应该随着你的设备的计 |

| | | |
|---|---|---|
| | 算能力的提升而增加 | 算能力的提升而增加 |

确保生产环境下已经设置了 `DEBUG = False` 。如果忘记关掉，用户会很乐意对你的服务器执行任意的Python代码。

# instance文件夹

有时你需要定义一些不能为人所知的配置变量。为此，你会想要把它们从*config.py*中的其他变量分离出来，并保持在版本控制之外。 你可能要隐藏类似数据库密码和API密钥的秘密，或定义特定于当前机器的参数。 为了让这更加轻松，Flask提供了一个叫*instance*文件夹的特性。 instance文件夹是根目录的一个子文件夹，包括了一个特定于当前应用实例的配置文件。我们不要把它提交到版本控制中。

这是一个使用了instance文件夹的简单Flask应用的结构:

```
config.py
requirements.txt
run.py
instance/
  config.py
yourapp/
  __init__.py
  models.py
  views.py
  templates/
  static/
```

## 使用instance文件夹

要想加载定义在instance文件夹中的配置变量，你可以使用 `app.config.from_pyfile()` 。如果在调用 `Flask()` 创建应用时设置了 `instance_relative_config=True` ，`app.config.` 将查看在*instance*文件夹的特殊文件。

```
app = Flask(__name__, instance_relative_confi
app.config.from_object('config')
app.config.from_pyfile('config.py')
```

现在，你可以在*instance/config.py*中定义变量，一如在*config.py*。 你也应该将instance文件夹加入到版本控制系统的忽略名单中。比如假设你用的是git，你需要在*gitignore*中新开一行，写下 `instance/` 。

## 密钥

instance文件夹的隐秘属性使得它成为藏匿密钥的好地方。 你可以在放入应用的密钥或第三方的API密钥。假如你的应用是开源的，或者将会是开源的，这会很重要。我们希望其他人去使用他们自己申请的密钥。

```
# instance/config.py

SECRET_KEY = 'Sm9obiBTY2hyb20ga2lja3MgYXNz'
STRIPE_API_KEY = 'SmFjb2IgS2FwbGFuLU1vc3MgaXMg
SQLALCHEMY_DATABASE_URI= \
"postgresql://user:TWljaGHFgiBCYXJ0b3N6a2lld2I
```

## 最小化依赖于环境的配置

如果你的生产环境和开发环境之间的差别非常小，你可以使用你的instance文件夹抹平配置上的差别。在*instance/config.py*中定义的变量可以覆盖在*config.py*中设定的值。你只需要在`app.config.from_object()`之后才调用`app.config.from_pyfile()`。这样做的其中一个优点是你可以在不同的机器中修改你的应用的配置。你的开发版本库可能看上去像这样:

config.py

```
DEBUG = False
SQLALCHEMY_ECHO = False
```

instance/config.py

```
DEBUG = True
SQLALCHEMY_ECHO = True
```

然后在生产环境中，你将这些代码从*instance/config.py*中移除，它就会改用回*config.py*中设定的变量。

> 参见
>
> - 在这里可以读到关于Flask-SQLAlchemy的配置密钥: http://pythonhosted.org/Flask-SQLAlchemy/config.html#configuration-keys

## 依照环境变量来配置

instance文件夹不应该在版本控制中。这意味这你将不能追踪你的instance配置。 在只有一两个变量的情况下这不是什么问题，但如果你有关于多个环境（生产，稳定，开发，等等）的一大堆配置，你不会愿意冒失去它们的风险。

Flask给我们提供了根据环境变量选择一个配置文件的能力。 这意味着我们可以在我们的版本库中有多个配置文件，并总是能根据具体环境，加载到对的那个。

当我们到了有多个配置文件共存的境况，是时候把文件都移动到 `config` 包之下。 下面是在这样的一个版本库中大致的样子:

```
requirements.txt
run.py
config/
  __init__.py # 空的，只是用来告诉Python它是一个包
  default.py
  production.py
  development.py
  staging.py
instance/
  config.py
yourapp/
  __init__.py
  models.py
  views.py
  static/
  templates/
```

在我们有一些不同的配置文件的情况下，可以这样设置:

| 文件名 | 内容 |
| --- | --- |
| config/default.py | 默认值，适用于所有的环或交由具体环境进行覆盖举个例子，在*config/default.py*中设置 `DEBUG = False` ，在*config/development.p*设置 `DEBUG = True` 。 |
| config/development.py | 在开发环境中用到的值。里你可以设定在localhos用到的数据库URI链接。 |
| config/production.py | 在生产环境中用到的值。里你可以设定数据库服务的URI链接，而不是开发境下的本地数据库URI链接。 |
| config/staging.py | 在你的开发过程中，你可需要在一个模拟生产环境服务器上测试你的应用。也许会使用不一样的数据库，想要为稳定版本的应替换掉一些配置。 |

要在不同的环境中指定所需的变量，你可以调用 `app.config.from_envvar()` :

```
# yourapp/__init__.py

app = Flask(__name__, instance_relative_confi
app.config.from_object('config.default')
app.config.from_pyfile('config.py') # 从instar
app.config.from_envvar('APP_CONFIG_FILE')
```

`app.config.from_envvar(‘APP_CONFIG_FILE’)` 将加载由环境变量 `APP_CONFIG_FILE` 指定的文件。这个环境变量的值应该是一个配置文件的绝对路径。

这个环境变量的设定方式取决于你运行你的应用的平台。如果你是在一台标准的Linux服务器上运行，你可以使用一个shell脚本来设置环境变量并运行 `run.py` 。

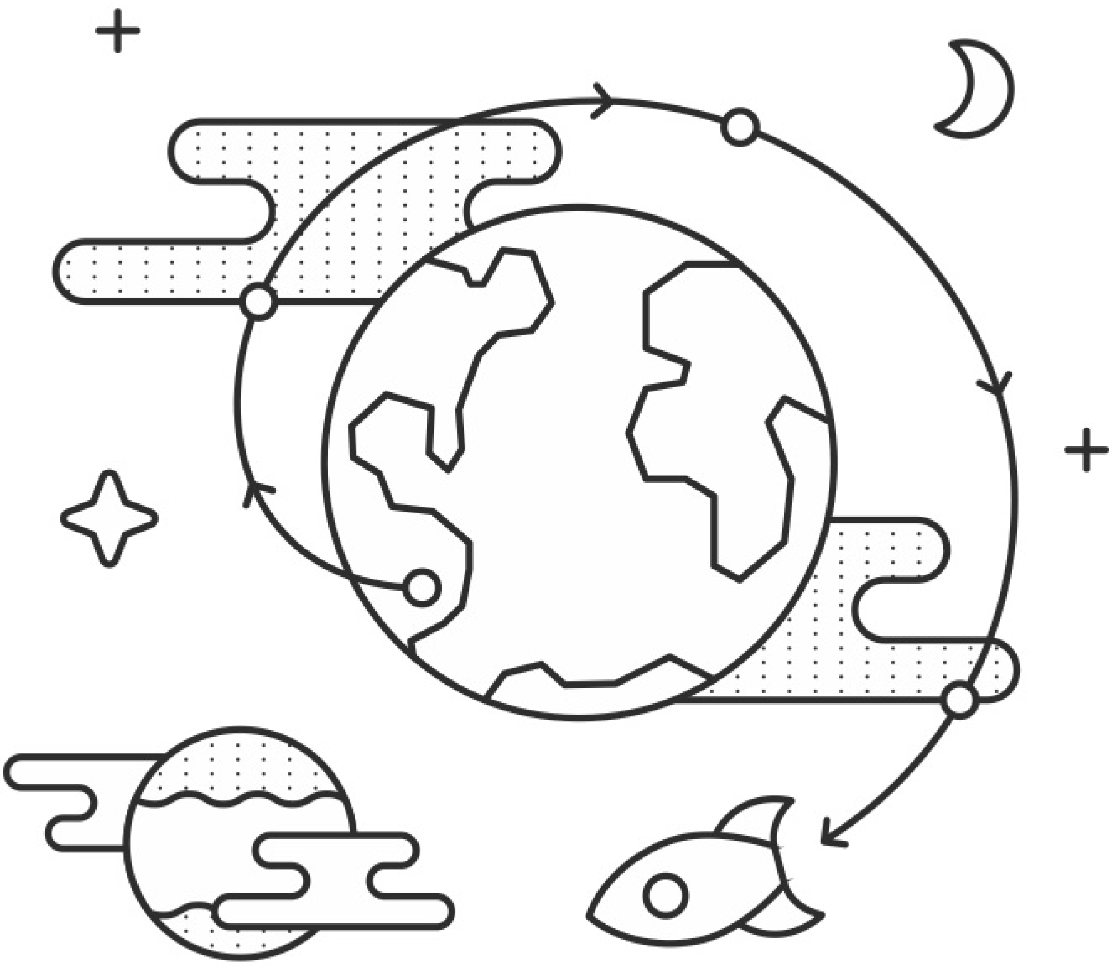
start.sh

```
APP_CONFIG_FILE=/var/www/yourapp/config/produ
python run.py
```

*start.sh*特定于某个环境，所以它也不能放入版本控制当中。如果你把应用托管到Heroku，你可以用Heroku提供的工具设置环境变量参数。对于其他PAAS平台也是同样的处理。

# 总结

- 一个简单的应用也许仅需一个配置文件:*config.py*
- instance文件夹可以帮助我们隐藏不愿为人所知的配置变量。
- instance文件夹可以用来改变特定环境下的程序配置。
- 应对复杂的，基于环境的配置，我们可以结合环境变量和`app.config.from_envvar()`来使用。



# 关于视图和路由的进阶技巧

## 视图装饰器

Python装饰器让我们可以用其他函数包装特定函数。 当一个函数被一个装饰器"装饰"时，那个装饰器会被调用，接着会做额外的工作，修改变量，调用原来的那个函数。我们可以把我们想要重用的代码作为装饰器来包装一系列视图。

装饰器的语法看上去像这样:

```
@decorator_function
def decorated():
    pass
```

如果你看过Flask入门指南，那么对这个语法应该不感到陌生。 `@app.route` 正是用于在Flask应用中给视图函数设定路由URL的装饰器。

让我们看一下在你的Flask应用中用得上的一些别的装饰器。

## 认证

Flask-Login使得用户认证系统的实现不再困难。除了处理用户认证的细节之外，Flask-Login允许我们使用 `@login_required` 这个装饰器来验证用户对某些资源的访问权限。

下面是从一个用到Flask-Login和 `@login_required` 装饰器的一个示范应用中获取的例子:

```
from flask import render_template
from flask_login import login_required, curre


@app.route('/')
def index():
    return render_template("index.html")

@app.route('/dashboard')
@login_required
def account():
    return render_template("account.html")
```

> 注意 `@app.route` 必须是最外面的视图装饰器。

只有已经验证的用户能够接触到*/dashboard*路由。你可以配置Flask-Login来重定向未验证用户到登录页面，返回HTTP 401状态码或别的你乐意的事。

> 参见 通过官方文档可以读到更多关于Flask-Login的内容

# 缓存

意淫一下，假如你的应用突然有一天在微博/朋友圈或网上别的地方火了。 于是秒秒钟会有成千上万的请求涌向你的应用。你的主页在每个请求中都要从数据库跑上一大趟，结果海量的请求导致网站慢得像教务系统一样。 你能做什么来加速这一过程，以免用户以为你的应用挂掉了？

答案不止一个，不过就本章主旨而言，标准答案是实现缓存。 特别的，我们将要用到Flask-Cache拓展。这个拓展给我们提供一个可以用来缓存某个响应一段时间的装饰器。

你可以将Flask-Cache配置成跟你想用的后台缓存一起使用。一个普遍的选择是Redis，一个容易配置和使用的软件。假设Flask-Cache已经配置好了，下面是我们的被装饰的视图的例子:

```
from flask_cache import Cache
from flask import Flask

app = Flask()

# 通过这个方式获取相关配置
cache = Cache(app)

@app.route('/')
@cache.cached(timeout=60)
def index():
    [...] # 进行一些数据库调用来获取所需信息
    return render_template(
        'index.html',
        latest_posts=latest_posts,
        recent_users=recent_users,
        recent_photos=recent_photos
    )
```

现在这个函数将会在每60秒最多运行一次。响应的结果会被保存在缓存中，并可以让期间的每一个请求获取。

> 注意 Flask-Cache同时允许我们记住函数 - 或缓存通过给定的参数调用的某个函数。你甚至可以缓存过于复杂的Jinja2模板片段!

## 自定义装饰器

在这个例子中，让我们假设我们有一个应用，每个月要求用户定期付费。如果一个用户的账户已经过期，我们要重定向他们到账单页面，并告知其悲伤的现实。

myapp/util.py

```
from functools import wraps
from datetime import datetime

from flask import flash, redirect, url_for

from flask_login import current_user

def check_expired(func):
    @wraps(func)
    def decorated_function(*args, **kwargs):
        if datetime.utcnow() > current_user.ac
            flash("Your account has expired. I
            return redirect(url_for('account_I
        return func(*args, **kwargs)

    return decorated_function
```

1. 当用 `@check_expired` 装饰一个函数时， `check_expired()` 被调用，被装饰的函数作为一个参数被传递进来。
2. `@wraps` 是一个装饰器，告知Python函数 `decorated_function()` 包装了视图函数 `func()` 。严格来说这不是必须的，但是这么做会使得装饰函数更加自然一些，更有利于文档和调试。
3. `decorated_function` 将截取原本传递给视图函数 `func()` 的args和kwargs。在这里我们检

查用户的账户是否过期。如果是，我们将闪烁一则信息，并重定向到账单页面。

4. 既然已经处理好自己的事情，我们把原来的参数交由视图函数 `func()` 去继续执行。

位于最顶部的装饰器将最先运行，然后调用下一个函数：一个视图函数或下一个装饰器。装饰器语法只是一个语法糖而已。

```
# 这样
@foo
@bar
def one():
    pass
```

```
# 等同于这样:
def two():
    pass
two = foo(bar(two))
r2 = two()

r1 == r2 # True
```

下面这个例子用到了我们自定义的装饰器和来自Flask-Cache拓展的`@login_required`装饰器。我们可以将多个装饰器堆成栈来一起使用。

myapp/views.py

```
from flask import render_template

from flask_login import login_required

from . import app
from .util import check_expired

@app.route('/use_app')
@login_required
@check_expired
def use_app():
    """欢迎光临"""

    return render_template('use_app.html')

@app.route('/account/billing')
@login_required
def account_billing():
    """拿账单来"""
    # [...]
    return render_template('account/billing.ht
```

当一个用户试图访问*/use_app*时，`check_expired()`将在执行视图函数之前确保相关的账户资料不会泄漏。

> 参见 在Python文档中可以读到更多关于`wraps()`的内容：http://docs.python.org/2/library/functools.html#functools.wraps

# URL转换器

## 内建转换器

当你在Flask中定义一个路由时，你可以将指定的一部分转换成Python变量并传递给视图函数。

```
@app.route('/user/<username>')
def profile(username):
    pass
```

在URL中作为的那一部分内容将作为 `username` 参数传递给视图函数。你也可以指定一个转换器过滤出特定的类型。

```
@app.route('/user/id/<int:user_id>')
def profile(user_id):
    pass
```

在这个代码块中，http://myapp.com/user/id/tomato 这个URL会返回一个404状态码 -- 此物无处觅。 这是因为URL中预期是整数的部分却遇到了一串字符串。

我们可以有另外一个接受一个字符串的视图函数。*/usr/id/tomato/*将调用它，而前一个函数只会被*/user/id/124*所调用。

下面是来自Flask文档的关于默认转换器的表格:

| 类型 | 作用 |
| --- | --- |
| string | 接受任何没有斜杠 `/` 的文本（默认） |
| int | 接受整数 |
| float | 类似于 `int`，但是接受的是浮点数 |
| path | 类似于 `string`，但是接受斜杠 `/` |

## 自定义转换器

我们也可以按照自己的需求打造自定义的转换器。 Reddit - 一个知名的链接分享网站 - 用户在此可以创建和管理基于主题和链接分享的社区。比如 `/r/python` 和 `/r/flask`，分别由URL `reddit.com/r/python` 和 `reddit.com/r/flask` 表示。 Reddit有一个有趣的特性是，通过在URL中用一个 `+` 隔开各个社区名，你可以同时看到来自多个社区的帖子。比如 `reddit.com/r/python+flask`。

我们可以使用一个自定义转换器来实现这种特性。 我们可以接受由加号隔离开来的任意数目参数，通过我们的ListConverter转换成一个列表，

并传递给视图函数。

util.py

```
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):

    def to_python(self, value):
        return value.split('+')

    def to_url(self, values):
        return '+'.join(BaseConverter.to_url(v
                        for value in values)
```

我们需要定义两个方法: `to_python()` 和 `to_url()` 。一如其名，`to_python()` 用于转换路径成一个Python对象，并传递给视图函数。而 `to_url()` 被 `url_for()` 调用，来转换参数成为符合URL的形式。

为了使用我们的ListConverter，我们首先得将它的存在告知Flask。

/myapp/__init__.py

```
from flask import Flask

app = Flask(__name__)

from .util import ListConverter

app.url_map.converters['list'] = ListConverte
```

> 注意 假如你的util模块有一行`from . import app`，那么有可能陷入循环import的问题。这就是为什么我等到app初始化之后才import ListConverter。

现在我们可以一如使用内建转换器一样使用我们的转换器。我们在字典中指定它的键为"list"，所以我们可以在`@app.route()`中这样使用:

views.py

```
from . import app

@app.route('/r/<list:subreddits>')
def subreddit_home(subreddits):
    """显示给定subreddits里的所有帖子"""
    posts = []
    for subreddit in subreddits:
        posts.extend(subreddit.posts)

    return render_template('/r/index.html', po
```

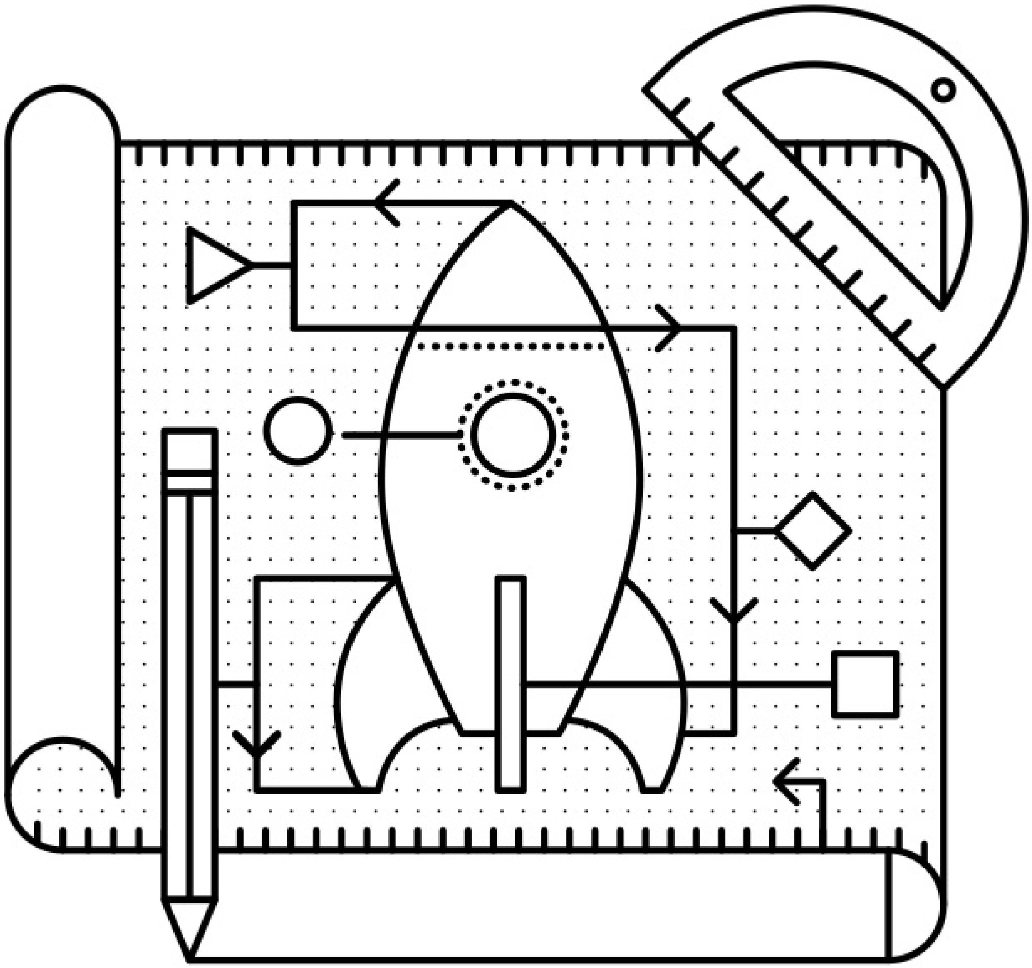
这应该会像Reddit的子社区系统一样工作。这样的方法可以用来实现你能想到的URL转换器。

# 总结

- Custom URL converters can be a great way to implement creative features involving URL's.
- 来自Flask-Login的 `@login_required` 装饰器可以帮助你限制验证用户对视图的访问。
- Flask-Cache插件为你提供一组装饰器来实现多种方式的缓存。
- 我们可以开发自定义视图装饰器来帮助我们

组织自己的代码，并坚守DRY(Don't Repeat Yourself 不重复你自己)原则。

- 自定义的URL转换器将会让你很嗨地玩转URL。



# 蓝图

## 什么是蓝图？

一个蓝图定义了可用于单个应用的视图，模板，静态文件等等的集合。举个例子，想象一下我们有一个用于管理面板的蓝图。这个蓝图将定义像*/admin/login*和*/admin/dashboard*这样的路由的视图。它可能还包括所需的模板和静态文件。你可以把这个蓝图当做你的应用的管理面板，管它是宇航员的交友网站，还是火箭推销员的CRM系统。

# 我什么时候会用到蓝图？

蓝图的杀手锏是将你的应用组织成不同的组件。假如我们有一个微博客，我们可能需要有一个蓝图用于网站页面，比如*index.html*和*about.html*。然后我们还需要一个用于在登录面板中展示最新消息的蓝图，以及另外一个用于管理员面板的蓝图。站点中每一个独立的区域也可以在代码上隔绝开来。最终你将能够把你的应用依据许多能完成单一任务的小应用组织起来。

> 参见 从Flask文档中读到更多使用蓝图的理由 Why Blueprints

# 我要把它们放哪里？

就像Flask里的每一件事情一样，你可以使用多种方式组织应用中的蓝图。对我而言，我喜欢按照功能(functional)而非分区(divisional)来组织。(这些术语是我从商业世界借来的)

## 功能式架构

在功能式架构中，按照每部分代码的功能来组织你的应用。所有模板放到同一个文件夹中，静态文件放在另一个文件夹中，而视图放在第三个文件夹中。

```
yourapp/
    __init__.py
    static/
    templates/
        home/
        control_panel/
        admin/
    views/
        __init__.py
        home.py
        control_panel.py
        admin.py
    models.py
```

除了*yourapp/views/__init__.py*，在*yourapp/views/*文件夹中的每一个*.py*文件都是一个蓝图。在*yourapp/__init__.py*中，我们将加载这些蓝图并在我们的`Flask()`对象中注册它们。等会我们将在本章了解到这是怎么实现的。

> 参见 当我下笔之时，flask.pocoo.org（Flask官网）就是使用这样的结构的。
> https://github.com/mitsuhiko/flask/tree/website/flask_website

## 分区式架构

在分区式架构中，按照每一部分所属的蓝图来组织你的应用。管理面板的所有的模板，视图和静态文件放在一个文件夹中，用户控制面板的则放在另一个文件夹中。

```
yourapp/
    __init__.py
    admin/
        __init__.py
        views.py
        static/
        templates/
    home/
        __init__.py
        views.py
        static/
        templates/
    control_panel/
        __init__.py
        views.py
        static/
        templates/
    models.py
```

在像上面列举的分区式结构，每一个*yourapp/*之下的文件夹都是一个独立的蓝图。所有的蓝图通过顶级的*__init__.py*注册到`Flask()`中。

# 哪种更胜一筹？

选择使用哪种架构实际上是一个个人问题。两者间的唯一区别是表达层次性的方式不同 -- 你可以使用任意一种方式架构Flask应用 -- 所以你所需的就是选择贴近你的需求的那个。

如果你的应用是由独立的，仅仅共享模型和配置的各组件组成，分区式将是个好选择。一个例子是允许用户建立网站的SaaS应用。你将会有独立的蓝图用于主页，控制面板，用户网站，和高亮面板。这些组件有着完全不同的静态文件和布局。如果你想要将你的蓝图提取成插件，或用之于别的项目，一个分区式架构将是正确的选择。

另一方面，如果你的应用的组件之间的联系较为紧密，使用功能式架构会更好。如果Facebook是用Flask开发的，它将有一系列蓝图，用于静态页面(比如登出主页，注册页面，关于，等等)，面板(比如最新消息)，用户内容(/robert/about和/robert/photos)，还有设置页面(/settings/security和/settings/privacy)以及别的。

这些组件都共享一个通用的布局和风格，但每一个都有它自己的布局。下面是一个非常精简的可能的Facebook结构，假定它用的是Flask。

```
facebook/
    __init__.py
    templates/
        layout.html
        home/
            layout.html
            index.html
            about.html
            signup.html
            login.html
        dashboard/
            layout.html
            news_feed.html
            welcome.html
            find_friends.html
        profile/
            layout.html
            timeline.html
            about.html
            photos.html
            friends.html
            edit.html
        settings/
            layout.html
            privacy.html
            security.html
            general.html
    views/
        __init__.py
        home.py
```

```
        dashboard.py
        profile.py
        settings.py
    static/
        style.css
        logo.png
    models.py
```

位于*facebook/view/*下的蓝图更多的是视图的集合而非独立的组件。同样的静态文件将被大多数蓝图重用。大多数模板都拓展自一个主模板。一个功能式的架构是组织这个项目的好的方式。

# 我该怎么使用它们？

## 基本用法

让我们看看来自Facebook例子的一个蓝图的代码：

facebook/views/profile.py

```
from flask import Blueprint, render_template

profile = Blueprint('profile', __name__)

@profile.route('/<user_url_slug>')
def timeline(user_url_slug):
    # 做些处理
    return render_template('profile/timeline.h

@profile.route('/<user_url_slug>/photos')
def photos(user_url_slug):
    # 做些处理
    return render_template('profile/photos.htm

@profile.route('/<user_url_slug>/about')
def about(user_url_slug):
    # 做些处理
    return render_template('profile/about.html
```

要想创建一个蓝图对象，你需要import `Blueprint()` 类并用参数 `name` 和 `import_name` 初始化。通常用 `__name__` ，一个表示当前模块的特殊的Python变量，作为 `import_name` 的取值。

假如使用分区式架构，你得告诉Flask某个蓝图是有着自己的模板和静态文件夹的。下面是这种情况下我们的定义大概的样子：

```
profile = Blueprint('profile', __name__,
                    template_folder='template
                    static_folder='static')
```

现在我们已经定义好了蓝图。是时候向Flask app注册它了。

facebook/__init__.py

```
from flask import Flask
from .views.profile import profile

app = Flask(__name__)
app.register_blueprint(profile)
```

现在在*fackbook/views/profile.py*中定义的路径(比如 `/<user_url_slug>` )会被注册到应用中，就像是被通过 `@app.route()` 定义的。

## 使用一个动态的URL前缀

继续看Facebook的例子，注意到所有的个人信息路由都以 `<user_url_slug>` 开头并把它传递给视图函数。我们想要用户通过类似*http://facebook.com/john.doe*的URL访问个人信息。通过给所有的蓝图的路由定义一个动态前缀，我们可以结束这种单调的重复。

蓝图允许我们定义静态的或动态的前缀。举个例子，我们可以告诉Flask蓝图中所有的路由应该以*/profile*作为前缀；这样是一个静态前缀。在Fackbook这个例子中，前缀取决于用户浏览的是谁的个人信息。他们在URL对应片段中输入的文本将决定我们输出的视图；这样是一个动态前缀。

我们可以选择何时定义我们的前缀。我们可以在下列两个时机中选择一个定义前缀：当我们实例化 `Blueprint()` 类的时候，或当我们在 `app.register_blueprint()` 中注册的时候。

下面我们在实例化的时候设置URL前缀：

facebook/views/profile.py

```
from flask import Blueprint, render_template

profile = Blueprint('profile', __name__, url_p

# [...]
```

下面我们在注册的时候设置URL前缀：

facebook/__init__.py

```
from flask import Flask
from .views.profile import profile

app = Flask(__name__)
app.register_blueprint(profile, url_prefix='/<
```

尽管这两种方式在技术上没有区别，最好还是在注册的同时定义前缀。这使得前缀的定义可以集中到顶级目录中。因此，我推荐使用 `url_prefix` 。

我们可以在前缀中使用转换器(converters)，就像调用route()一样。同样也可以使用我们定义过的任意自定义转换器。通过这样做，我们可以自动

处理在蓝图前缀中传递过来的值。在这个例子中，我们将根据URL片段获取用户类并传递到我们的profile蓝图中。我们将通过一个名为`url_value_preprocessor()`装饰器来做到这一点。

facebook/views/profile.py

```
from flask import Blueprint, render_template,

from ..models import User

# The prefix is defined in facebook/__init__.p
profile = Blueprint('profile', __name__)

@profile.url_value_preprocessor
def get_profile_owner(endpoint, values):
    query = User.query.filter_by(url_slug=valu
    g.profile_owner = query.first_or_404()

@profile.route('/')
def timeline():
    return render_template('profile/timeline.h

@profile.route('/photos')
def photos():
    return render_template('profile/photos.htm

@profile.route('/about')
def about():
    return render_template('profile/about.html
```

我们使用 `g` 对象来储存个人信息的拥有者，而g可以用于Jinja2模板上下文。这意味着在这个简单的例子中，我们仅仅需要渲染模板，需要的信息就能在模板中获取。

facebook/templates/profile/photos.html

```
{% extends "profile/layout.html" %}

{% for photo in g.profile_owner.photos.all() %
    <img src="{{ photo.source_url }}" alt="{{
{% endfor %}
```

> 参见 Flask文档中有一个关于如何将你的URL国际化的好教程：
> http://flask.pocoo.org/docs/patterns/urlprocessors/#internationalized-blueprint-urls }

## 使用一个动态子域名

今天，许多SaaS应用提供用户一个子域名来访问他们的软件。举个例子，Harvest，是一个针对顾问的日程管理软件，它在yourname.harvestapp.com给你提供了一个控制面板。下面我将展示在Flask中如何像这样自动生成一个子域名。

在这一节，我将使用一个允许用户创建自己的网站的应用作为例子。假设我们的应用有三个蓝图分别针对以下的部分：用户注册的主页面，可用于建立自己的网站的用户管理面板，用户的网站。考虑到这三个部分相对独立，我们将用分区式结构组织起来。

```
sitemaker/
    __init__.py
    home/
        __init__.py
        views.py
        templates/
            home/
        static/
            home/
    dash/
        __init__.py
        views.py
        templates/
            dash/
        static/
            dash/
    site/
        __init__.py
        views.py
        templates/
            site/
        static/
            site/
    models.py
```

| url | 蓝图目录 |
|---|---|
| sitemaker.com/ | sitemaker/home |
| bigdaddy.sitemaker.com | sitemaker/site |
| bigdaddy.sitemaker.com/admin | sitemaker/dash |

定义动态子域名的方式和定义URL前缀一样。同样的，我们可以选择在蓝图文件夹中，或在顶级目录的__init__.py中定义它。这一次，我们还是在*sitemaker/__init__.py*中放置所有的定义。

sitemaker/__init__.py

```
from flask import Flask
from .site import site

app = Flask(__name__)
app.register_blueprint(site, subdomain='<site_
```

既然我们用的是分区式架构，蓝图将在*sitemaker/site/__init__.py*定义。

sitemaker/site/__init__py

```
from flask import Blueprint

from ..models import Site

# 注意首字母大写的Site和全小写的site是两个完全不同的
# Site是一个模块，而site是一个蓝图。

site = Blueprint('site', __name__)

@site.url_value_preprocessor
def get_site(endpoint, values):
    query = Site.query.filter_by(subdomain=va
    g.site = query.first_or_404()

# 在定义site后才import views。视图模块需要import
from . import views
```

现在我们已经从数据库中获取可以向请求子域名的用户展示的站点信息了。

为了使Flask能够支持子域名，你需要修改配置变量 `SERVER_NAME` 。

*config.py*

```
SERVER_NAME = 'sitemaker.com'
```

> 注意 几分钟之前，当我正在打这一章的草稿时，聊天室中某人求助称他们的子域名能够在开发环境下正常工作，但在生产环境下就会失败。我问他们是否配置了`SERVER_NAME`，结果发现他们只在开发环境中配置了这个变量。在生产环境中设置这个变量解决了他们的问题。从这里可以看到我(imrobert)和aplavin之间的对话：
> http://dev.pocoo.org/irclogs/%23pocoo.2013-07-30.log
>
> 注意 你可以同时设置一个子域名和URL前缀。想一下使用上面的表格的URL结构，我们要怎样来配置*sitemaker/dash*。

# 使用蓝图重构小型应用

我打算通过一个简单的例子来展示用蓝图重写一个应用的几个步骤。我们将从一个典型的Flask应用起步，然后重构它。

```
config.txt
requirements.txt
run.py
gnizama/
  __init__.py
  views.py
  models.py
  templates/
  static/
tests/
```

*views.py*文件已经膨胀到10,000行代码了。重构的工作被一推再推，到现在已经无路可退。这个文件包括了我们的网站的所有的视图，比如主页，用户面板，管理员面板，API和公司博客。

## Step 1：分区式还是功能式？

这个应用由关联较小的各部分构成。模板和静态文件不太可能在蓝图间共享，所以我们将使用分区式结构。

# Step 2：分而治之

> 注意 在你对你的应用大刀阔斧之前，把一切提交到版本控制。你不会接受对任何有用的东西的意外删除。

接下来我们将继续前进，为我们的新应用创建目录树。从为每一个蓝图创建一个目录开始吧。然后整体复制*views.py*，*static/*和*templates/*到每一个蓝图文件夹。接着你可以从顶级目录删除掉它们了。

```
config.txt
requirements.txt
run.py
gnizama/
  __init__.py
  home/
    views.py
    static/
    templates/
  dash/
    views.py
    static/
    templates/
  admin/
    views.py
    static/
    templates/
  api/
    views.py
    static/
    templates/
  blog/
    views.py
    static/
    templates/
  models.py
tests/
```

## Step 3：大扫除

现在我们可以到每一个蓝图中，移除无关的视图，静态文件和模板。你在这一阶段的处境很大程度上取决于一开始你是怎么组织你的应用的。

最终结果应该是：每个蓝图有一个 `views.py` 包括了蓝图里的所有视图，没有两个蓝图对同一个路由定义了视图；每一个*templates/*文件夹应该只包括该蓝图所需的模板；每一个*static/*文件夹应该只包括该蓝图所需的静态文件。

> 注意 趁此机会消除所有不必要的import。很容易忽略掉他们的存在，但他们会拥塞你的代码，甚至拖慢你的应用。

# Step 4：蓝图

在这一部分我们把文件夹转换成蓝图。关键在于*__init__.py*文件。作为开始，让我们看一下API蓝图的定义。

*gnizama/api/__init__.py*

```
from flask import Blueprint

api = Blueprint(
    'site',
    __name__,
    template_folder='templates',
    static_folder='static'
)

from . import views
```

接着我们可以在gnizama的顶级目录下的*__init__.py*中注册这个蓝图。

*gnizama/__init__.py*

```
from flask import Flask
from .api import api

app = Flask(__name__)

# 在api.gnizama.com中添加API蓝图
app.register_blueprint(api, subdomain='api')
```

确保路由现在是在蓝图中注册的，而不是在app对象。下面是在我们重构应用之前，一个在*gnizama/views.py*的API路由可能的样子。

*gnizama/views.py*

```
from . import app

@app.route('/search', subdomain='api')
def api_search():
    pass
```

在蓝图中它看上去像这样：

*gnizama/api/views.py*

```
from . import api

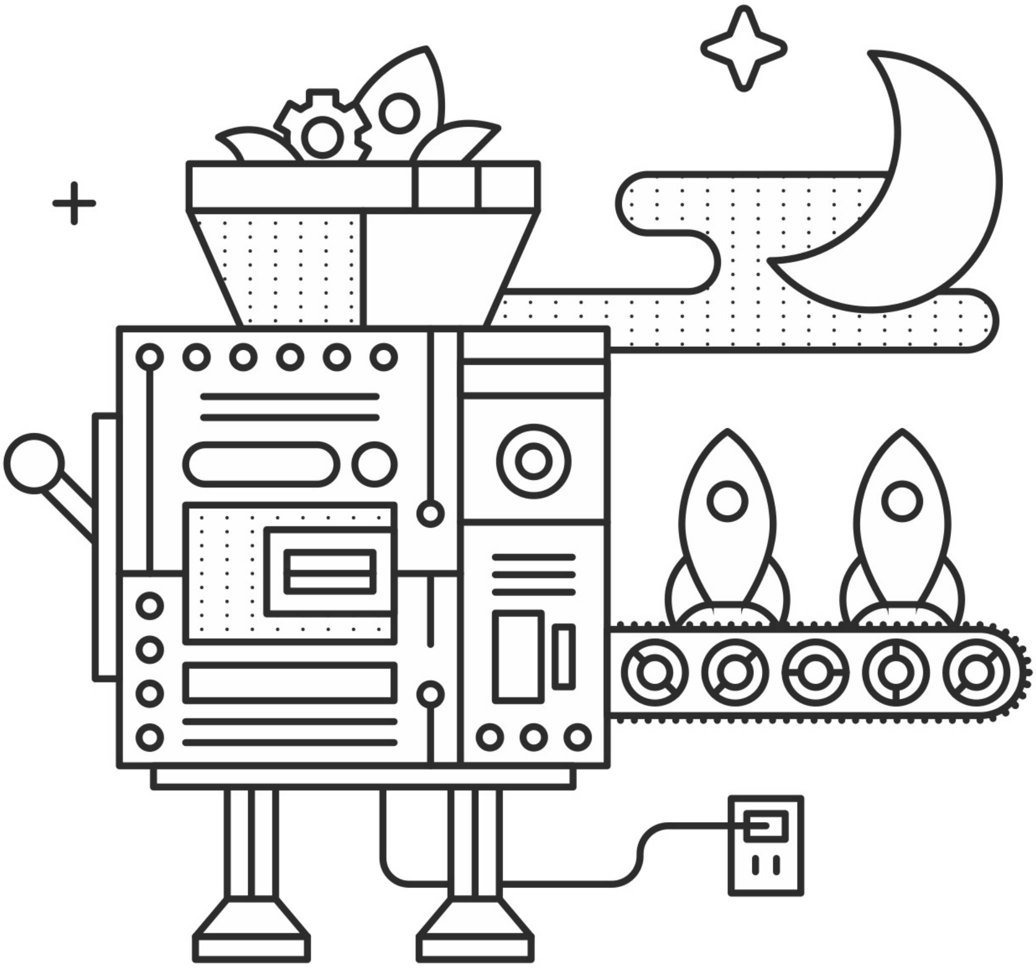
@api.route('/search')
def search():
    pass
```

## Step 5：大功告成

现在我们的应用已经比只有单个臃肿的*views.py*的时候更加模块化了。

# 总结

- 一个蓝图包括了可以作为独立应用的视图，模板，静态文件和其他插件。
- 蓝图是组织你的应用的好办法。
- 在分区式架构下，每个蓝图对应你的应用的一个部分。
- 在功能式架构下，每个蓝图就只是视图的集合。所有的模板和静态文件都放在一块。
- 要使用蓝图，你需要定义它，并在应用中用 `Flask.register_blueprint()` 注册它。
- 你可以给一个蓝图中的所有路由定义一个动态URL前缀。
- 你也可以给蓝图中的所有路由定义一个动态子域名。
- 仅需五步走，你可以用蓝图重构一个应用。



# 模板

尽管Flask并不强迫你使用某个特定的模板语言，它还是默认你会使用Jinja。在Flask社区的大多数开发者使用Jinja，并且我建议你也跟着做。有一

些插件允许你用其他模板语言进行替代(比如Flask-Genshi和Flask-Mako)，但除非你有充分理由（不懂Jinja可不是一个充分的理由！），否则请保持那个默认的选项；这样你会避免浪费很多时间来焦头烂额。

> 注意 几乎所有提及Jinja的资源讲的都是Jinja2。Jinja1确实曾存在过，但在这里我们不会讲到它。当你看到Jinja时，我们讨论的是这个Jinja: http://jinja.pocoo.org/

# Jinja快速入门

**Jinja**文档在解释这门语言的语法和特性这方面做得很棒。在这里我不会啰嗦一遍，但还是会再一次向你强调下面一点：

> Jinja有两种定界符。`{% ... %}`和`{{ ... }}`。前者用于执行像for循环或赋值等语句，后者向模板输出一个表达式的结果。
>
> 参见：http://jinja.pocoo.org/docs/templates/#synopsis

# 怎样组织模板

所以要将模板放进我们的应用的哪里呢？如果你是从头开始阅读的本文，你可能注意到了Flask在对待你如何组织项目结构的事情上十分随意。模板也不例外。你大概也已经注意到，总会有一个放置文件的推荐位置。记住两点。对于模板，这个最佳位置是放在包文件夹下。

```
myapp/
    __init__.py
    models.py
    views/
    templates/
    static/
run.py
requirements.txt
```

让我们打开模板文件夹看看。

```
templates/
    layout.html
    index.html
    about.html
    profile/
        layout.html
        index.html
    photos.html
    admin/
        layout.html
        index.html
        analytics.html
```

模板的结构平行于对应的路由的结构。对应于路由*myapp.com/admin/analytics*的模板是*templates/admin/analytics.html*。这里也有一些

额外的模板不会被直接渲染。*layout.html*文件就是用于被其他模板继承的。

## 继承

就像蝙蝠侠一样，一个组织良好的模板文件夹也离不开继承带来的好处。基础模板通常定义了一个适用于所有的子模板的主体结构。在我们的例子里，*layout.html*是一个基础模板，而其他的*html*文件都是子模板。

通常，你会有一个顶级的*layout.html*定义你的应用的主体布局，外加站点的每一个节点也有自己的一个*layout.html*。如果再看一眼上面的文件夹结构，你会看到一个顶级的*myapp/templates/layout.html*，以及*myapp/templates/profile/layout.html*和*myapp/templates/admin/layout.html*。后两个文件继承并修改第一个文件。

继承是通过 `{% extends %}` 和 `{% block %}` 标签实现的。在双亲模板中，你可以定义要给子模板处理的block。

*myapp/templates/layout.html*

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>{% block title %}{% endblock %
    </head>
    <body>
    {% block body %}
        <h1>这个标题在双亲模板中定义</h1>
    {% endblock %}
    </body>
</html>
```

在子模板中，你可以拓展双亲模板并定义block里面的内容。

*myapp/templates/index.html*

```
{% extends "layout.html" %}
{% block title %}Hello world!{% endblock %}
{% block body %}
    {{ super() }}
    <h2>这个标题在子模板中定义</h2>
{% endblock %}
```

`super()` 函数让我们在子模板里加载双亲模板中这个block的内容。

> 参见 若想了解更多关于继承的内容，请移步到Jinja模板继承方面的文档。
> http://jinja.pocoo.org/docs/templates/#template-inheritance

# 创建宏

凭借将反复出现的代码片段抽象成宏，我们可以实现DRY原则（Don't Repeat Yourself）。在撰写用于应用的导航功能的HTML时，我们可能会需要给“活跃”链接（比如，到当前页面的链接）一个不同的类。如果没有宏，我们将不得不使用一大堆if/else语句来从每个链接中过滤出“活跃”链接。

宏提供了模块化模板代码的一种方式；它们就像是函数一样。让我们看一下如何使用宏来标记活跃链接。

myapp/templates/layout.html

```
{% from "macros.html" import nav_link with co
<!DOCTYPE html>
<html lang="en">
    <head>
    {% block head %}
        <title>我的应用</title>
    {% endblock %}
    </head>
    <body>
        <ul class="nav-list">
            {{ nav_link('home', 'Home') }}
            {{ nav_link('about', 'About') }}
            {{ nav_link('contact', 'Get in tou
        </ul>
    {% block body %}
    {% endblock %}
    </body>
</html>
```

现在我们调用了一个尚未定义的宏 - `nav_link` - 并传递两个参数给它：一个目标（比如目标视图的函数名）和我们想要展示的文本。

> 注意 你可能注意到了我们在import语句中加入了**with context**。Jinja的上下文**(context)**包括了通过 `render_template()` 函数传递的参数以及在我们的Python代码的Jinja环境上下文。这些变量能够被用于模板的渲染。
>
> 一些变量是我们显式传递过去的，比如 `render_template("index.html", color="red")`，但还有些变量和函数是Flask自动加入到上下文的，比如 `request`，`g` 和 `session`。使用了 `{% from ... import ... with context %}`，我们告诉Jinja让所有的变量也在宏里可用。
>
> 参见
>
> - 所有的全局变量都是由Flask传递给Jinja上下文的: http://flask.pocoo.org/docs/templating/#standard-context
> - 通过上下文处理器（context processors），我们可以增加传递给Jinja上下文的变量和函数:

> [http://flask.pocoo.org/docs/templating/#context-processors](http://flask.pocoo.org/docs/templating/#context-processors)

是时候定义模板中用的 `nav_link` 宏了。

myapp/templates/macros.html

```
{% macro nav_link(endpoint, text) %}
{% if request.endpoint.endswith(endpoint) %}
    <li class="active"><a href="{{ url_for(end
{% else %}
    <li><a href="{{ url_for(endpoint) }}">{{te
{% endif %}
{% endmacro %}
```

现在我们已经在*myapp/templates/macros.html*中定义了一个宏。我们所做的，就是使用Flask的 `request` 对象 - 默认在Jinja上下文中可用 - 来检查当前路由是否是传递给 `nav_link` 的那个路由参数。如果是，我们就在目标链接指向的页面上，于是可以标记它为活跃的。

> 注意 `from x import y` 语句中要求x是相对于y的相对路径。如果我们的模板位于*myapp/templates/user/blog.html*，我们需要使用 `from "../macros.html" import nav_link with` 。

# 自定义过滤器

Jinja过滤器是在渲染成模板之前，作用于 `{{ ... }}` 中的表达式的值的函数。

```
<h2>{{ article.title|title }}</h2>
```

在这个代码中， `title` 过滤器接受 `article.title` 并返回一个标题格式的文本，用于输出到模板中。它的语法，以及功能，皆一如Unix中修改程序输出的“管道”一样。

> 参见 除了 `title` ，还有许许多多别的内建的过滤器。在这里可以看到完整的列表：http://jinja.pocoo.org/docs/templates/#builtin-filters

我们可以自定义用于Jinja模板的过滤器。作为例子，我们将实现一个简单的 `caps` 过滤器来使字符串中所有的字母大写。

> 注意 Jinja已经有一个 `upper` 过滤器能实现这一点，还有一个 `capitalize` 过滤器能大写第一个字符并小写剩余字符。这些过滤器还能处理Unicode转换，不过我们的这个例子将只专注于阐述相关概念。

我们将在*myapp/util/filters.py*中定义我们的过滤器。这个 `util` 包可以用来放置各种杂项。

myapp/util/filters.py

```
from .. import app

@app.template_filter()
def caps(text):
    """Convert a string to all caps."""
    return text.uppercase()
```

在上面的代码中，通过 `@app.template_filter()` 装饰器，我们能将某个函数注册成Jinja过滤器。默认的过滤器名字就是函数的名字，但是通过传递一个参数给装饰器，你可以改变它：

```
@app.template_filter('make_caps')
def caps(text):
    """Convert a string to all caps."""
    return text.uppercase()
```

现在我们可以在模板中调用 `make_caps` 而不是 `caps`：`{{ "hello world!"|make_caps }}`。

为了让我们的过滤器在模板中可用，我们仅需要在顶级_*_init_*_.py中import它。

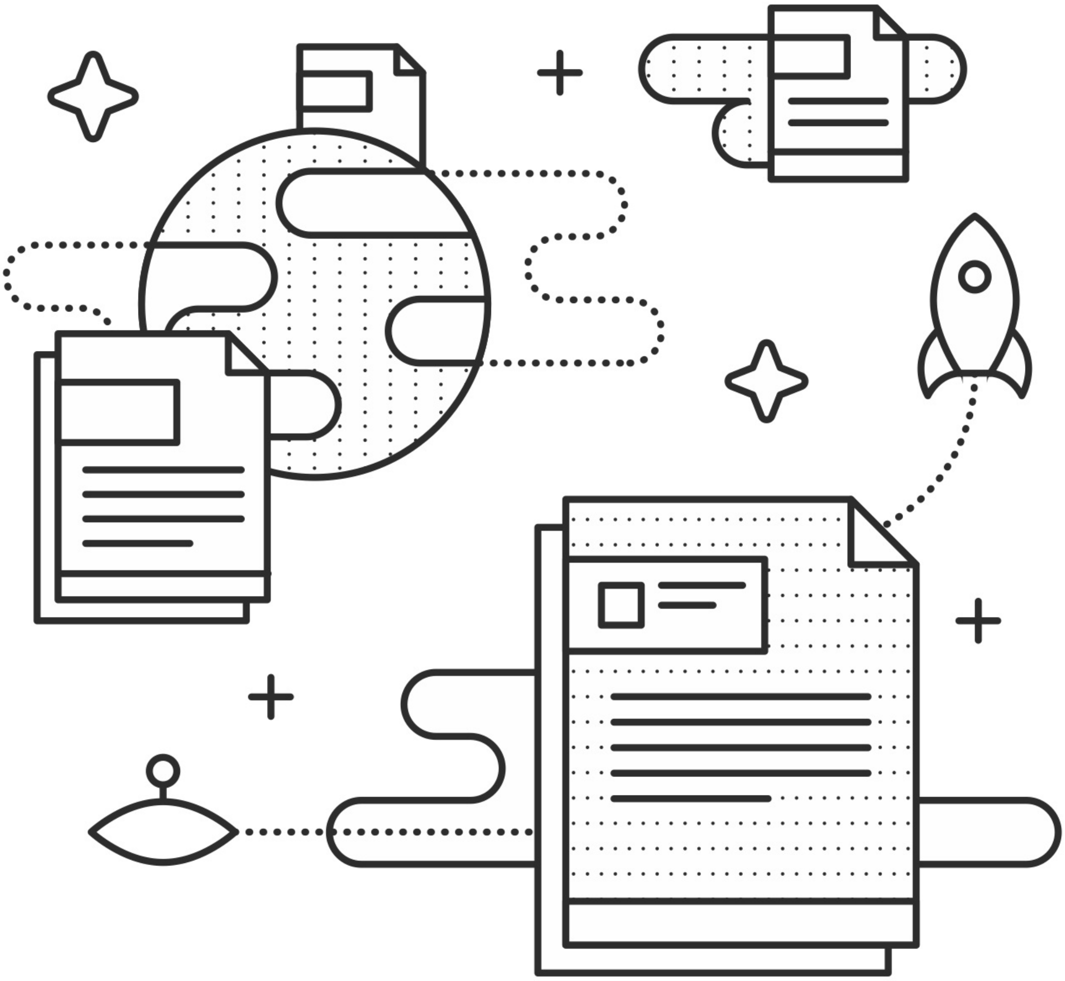
myapp/__init__.py

```
# 确保app已经被初始化以免导致循环import
from .util import filters
```

# 总结

- 使用Jinja作为模板语言。
- Jinja有两种定界符：`{% ... %}`和`{{ ... }}`。前者用于执行类似循环或赋值的语句，后者向模板输出表达式求值的结果。
- 模板应该放在*myapp/templates/* - 一个在应用文件夹里面的目录。
- 我建议*template/*文件夹的结构应该与应用URL结构一一对应。
- 你应该在*myapp/templates*以及站点的每一部分放置一个*layout.html*作为布局模板。后者是前者的拓展。
- 可以用模板语言写类似于函数的宏。
- 可以用Python代码写应用在模板中的过滤器

函数。



# 静态文件

一如其名，静态文件是那些不会改变的文件。一般情况下，在你的应用中，这包括CSS文件，Javascript文件和图片。它也可以包括视频文件和

其他可能的东西。

# 组织你的静态文件

我们将在应用的包中创建一个叫*static*的文件夹放置我们的静态文件。

```
myapp/
    __init__.py
    static/
    templates/
    views/
    models.py
run.py
```

*static/*里面的文件组织方式取决于个人的爱好。就我个人来说，如果第三方库（比如jQuery, Bootstrap等等）跟自己的Javascript和CSS文件混起来，我会因此而不爽。所以，我要将第三方库全放到一个*lib/*文件夹中。有时会用*vendor/*来代替*lib/*。

```
static/
    css/
        lib/
            bootstrap.css
        style.css
        home.css
        admin.css
    js/
        lib/
            jquery.js
        home.js
        admin.js
    img/
        logo.svg
        favicon.ico
```

# 提供一个favicon

用户将通过yourapp.com/static/访问你的静态文件夹中的文件。默认下浏览器和其他软件认为你的favicon位于yourapp.com/favicon.ico。要想解决这种不一致。你可以在站点模板的 `<head>` 部分添加下面内容。

```
<link rel="shortcut icon" href="{{ url_for('st
```

# 用Flask-Assets管理静态文件

Flask-Assets是一个管理静态文件的插件。它提供了两种非常有用的特性。首先，它允许你在Python代码中定义多组（bundles）可以同时插入你的模板的静态文件。其次，它允许你预处理这些文件。这意味着你可以合并并压缩你的CSS和Javascript文件，这样用户就会仅仅得到两个压缩后的文件（CSS和Javascript）而免于花费太多带宽。你甚至可以从Sass，Less，CoffeeScript或别的源码里编译出最终产物。

下面是这一章中也做例子的静态文件夹的基本结构。

*myapp/static/*

```
static/
    css/
        lib/
            reset.css
        common.css
        home.css
        admin.css
    js/
        lib/
            jquery-1.10.2.js
            Chart.js
        home.js
        admin.js
    img/
        logo.svg
        favicon.ico
```

# 定义分组

我们的应用有两部分：公共网站和管理面板（分别称作"home"和"admin"）。我们将定义四个分组来覆盖它：每个部分有一个Javascript和一个CSS分组。我们将它们放入 `util` 包里的assets模块。

*myapp/util/assets.py*

```
from flask_assets import Bundle, Environment
from .. import app

bundles = {

    'home_js': Bundle(
        'js/lib/jquery-1.10.2.js',
        'js/home.js',
        output='gen/home.js),

    'home_css': Bundle(
        'css/lib/reset.css',
        'css/common.css',
        'css/home.css',
        output='gen/home.css),

    'admin_js': Bundle(
        'js/lib/jquery-1.10.2.js',
        'js/lib/Chart.js',
        'js/admin.js',
        output='gen/admin.js),

    'admin_css': Bundle(
        'css/lib/reset.css',
        'css/common.css',
        'css/admin.css',
        output='gen/admin.css)
}

assets = Environment(app)

assets.register(bundles)
```

Flask-Assets按照被列出来的顺序合并你的文件。

如果*admin.js*依赖*jquery-1.10.2.js*，确保jquery被列在前面。

我们通过字典来定义分组，这样方便注册它们。webassets，实际上是Flask-Assets的核心，提供了一系列方式来注册分组，包括上面我们演示的以字典作参数的方法。（译注：webassets之于Flask-Assets，正如SQLAlchemy之于Flask-SQLAlchemy。）

> 参见 webassets在这里注册了分组：https://github.com/miracle2k/webassets/blob/0.8/src/webassets/env.py#L380

既然我们已经在 `util.assets` 中注册了我们的分组，剩下的就是在__init__.py中，在app对象初始化之后，来导入这个模块。

myapp/__init__.py

```
# [...] Initialize the app

from .util import assets
```

# 使用你的分组

下面是我们的例子中的模板文件夹：

*myapp/templates/*

```
templates/
    home/
        layout.html
        index.html
        about.html
    admin/
        layout.html
        dash.html
        stats.html
```

要使用我们的admin分组，我们将插入它们到admin部分的基础模板 - *admin/layout.html* - 中。

myapp/templates/admin/layout.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        {% assets "admin_js" %}
            <script type="text/javascript" src
        {% endassets %}
        {% assets "admin_css" %}
            <link rel="stylesheet" href="{{ AS
        {% endassets %}
    </head>
    <body>
    {% block body %}
    {% endblock %}
    </body>
</html>
```

对于home分组，我们也同样在*templates/home/layout.html*做一样的处理。

# 使用过滤器

我们可以使用webassets过滤器来预处理我们的静态文件。这将方便我们压缩Javascript和CSS文件。现在修改下我们的代码来实现这一点。

myapp/util/assets.py

```
# [...]

bundles = {

    'home_js': Bundle(
        'lib/jquery-1.10.2.js',
        'js/home.js',
        output='gen/home.js',
        filters='jsmin'),

    'home_css': Bundle(
        'lib/reset.css',
        'css/common.css',
        'css/home.css',
        output='gen/home.css',
        filters='cssmin'),

    'admin_js': Bundle(
        'lib/jquery-1.10.2.js',
        'lib/Chart.js',
        'js/admin.js',
        output='gen/admin.js',
        filters='jsmin'),

    'admin_css': Bundle(
        'lib/reset.css',
        'css/common.css',
        'css/admin.css',
        output='gen/admin.css',
        filters='cssmin')
}

# [...]
```

> 注意 要想使用 `jsmin` 和 `cssmin` 过滤器，你需要安装jsmin和cssmin包（使用 `pip install jsmin cssmin` ）。确保把它们也加入到*requirements.txt*。

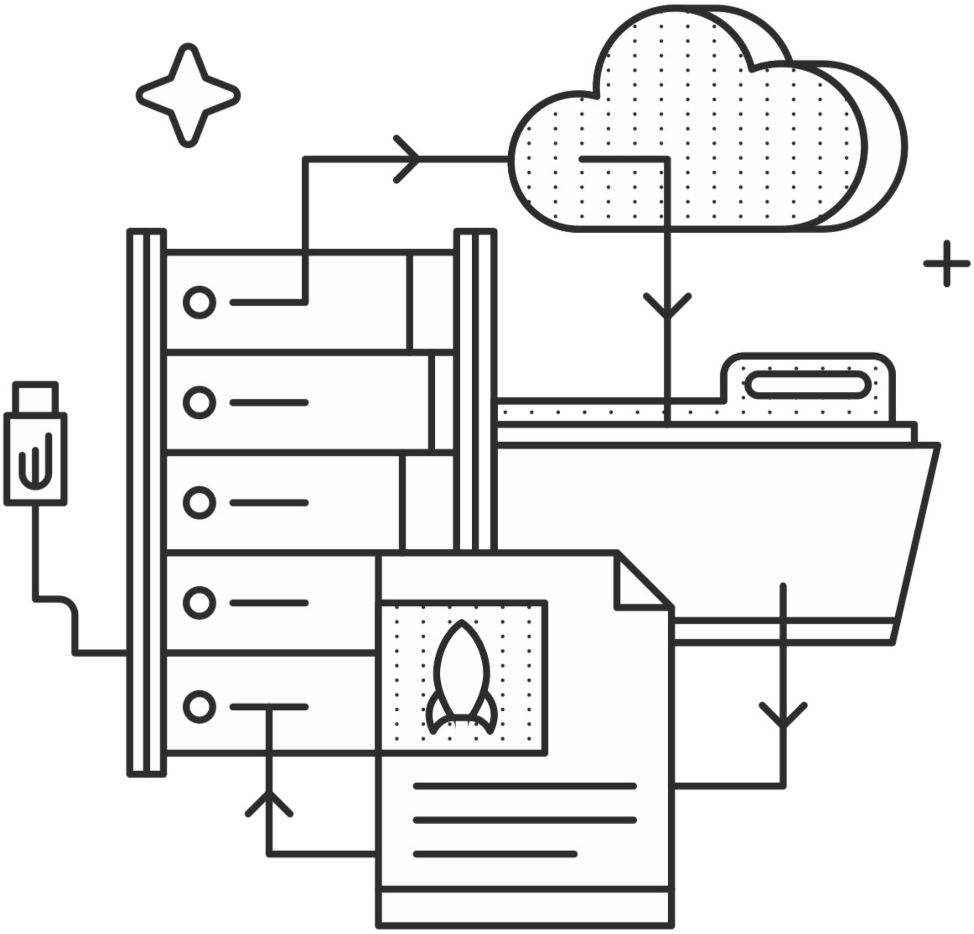
一旦模板已经渲染好，Flask-Assets将在合并的同时压缩我们的文件，而且当其中一个源文件改变时，它会自动更新压缩文件。

> 注意 如果你在配置中设置 `ASSETS_DEBUG = True` ，Flask-Assets将独立输出每一个源文件而不会合并它们。
>
> 参见 你可以使用Flask-Assets过滤器来自动编译Sass，Less，CoffeeScript，和其他预处理器。来看下你还可以使用哪些过滤器：http://elsdoerfer.name/docs/webassets/builtin_filters.html#js-css-compilers

# 总结

- 静态文件归于*static/*文件夹。

- 将第三方库跟你自己的静态文件隔离开来。
- 在你的模板里指定你的favicon的路径。
- 使用Flask-Assets来在模板插入你的静态文件。
- Flask-Assets可以编译，合并以及压缩你的静态文件。



# 存储

大多数Flask应用都将要跟数据打交道。有很多种不同的方法存储数据。至于哪种最优，取决于数据的类型。如果你储存的是关系性数据（比如一

个用户有多个邮件，一个邮件对应一个用户），关系型数据库无疑是你的选择。其他类型的数据也许适合储存到NoSQL数据库（比如MongoDB）中。

我不会告诉你如何为你的应用选择数据库。如果有人告诉你，NoSQL是你的唯一选择；那么必然也会有人建议用关系型数据库处理同样的问题。对此我唯一需要说的是，如果你不清楚，关系型数据库（MySQL, PostgreSQL等等）将满足你绝大部分的需求。

另外，当你使用关系型数据库，你就能用到SQLAlchemy，而SQLAlchemy用起来真爽。

# SQLAlchemy

SQLAlchemy是一个ORM（对象关系映射）。基于对目标数据库的原生SQL的抽象，它提供了与一长串数据库引擎的一致的API。这一列表中包括MySQL，PostgreSQL，和SQLite。这使得在你的

模型和数据库间交换数据变得轻松愉快，同时也使得诸如换掉数据库引擎和迁移数据库模式等其他事情变得没那么繁琐。

存在一个很棒的Flask插件使得在Flask中使用SQLAlchemy更为轻松。它就是Flask-SQLAlchemy。Flask-SQLAlchemy为SQLAlchemy设置了许多合理的配置。它也内置了一些session管理，这样你就不用在应用代码里处理这种基础事务了。

让我们深入看看一些代码。我们将先定义一些模型，接着配置下SQLAchemy。这些模型将位于*myapp/models.py*，不过首先我们要在*myapp/__init__.py*定义我们的数据库。

*myapp/__init__.py_*

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__, instance_relative_confi

app.config.from_object('config')
app.config.from_pyfile('config.py')

db = SQLAlchemy(app)
```

我们首先初始化并配置你的Flask应用，然后用它来初始化你的SQLAlchemy数据库处理程序。我们将为数据库配置使用一个instance文件夹，所以我们应该在初始化应用时加上 `instance_relative_config` 选项，然后调用 `app.config.from_pyfile` 。现在我们可以定义模型了。

*myapp/models.py*

```
from . import db

class Engine(db.Model):

    # Columns

    id = db.Column(db.Integer, primary_key=Tru

    title = db.Column(db.String(128))

    thrust = db.Column(db.Integer, default=0)
```

`Column` ，`Integer` ，`String` ，`Model` 和其他的SQLAlchemy类都可以通过由Flask-SQLAlchemy构造的 `db` 对象访问。我们会定义一个储存我们的太空飞船引擎的当前状态的模型。每个引擎有一个ID，一个标题和一个推力等级。

我们需要往我们的配置添加一些数据库信息。我们打算使用一个instance文件夹来避免配置变量被记录进版本控制系统，所以我们要把它们放入*instance/config.py*。

*instance/config.py*

```
SQLALCHEMY_DATABASE_URI = "postgresql://user:
```

> 注意 你的数据库URI将取决于你选择的数据库和它部署的位置。看一下这个相关的SQLAlchemy文档：http://docs.sqlalchemy.org/en/latest/core/engines.html?highlight=database#database-urls

# 初始化数据库

既然数据库已经配置好了，而模型也定义了，是时候初始化数据库了。这个步骤从由模型定义中创建数据库模式开始。

通常这会是非常痛苦的过程。不过幸运的是，SQLAlchemy提供了一个十分酷的工具帮我们完成了所有的琐事。

让我们在版本库的根目录下打开一个Python终端。

```
$ pwd
/Users/me/Code/myapp
$ workon myapp
(myapp)$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-50
Type "help", "copyright", "credits" or "licens
>>> from myapp import db
>>> db.create_all()
>>>
```

现在，感谢SQLAlchemy，你会发现在你配置的数据库中，所需的表格已经被创建出来了。

# Alembic迁移工具

数据库的模式并非亘古不变的。举个例子，你可能需要在引擎的表里添加一个 `last_fired` 的项。如果这个表是一张白纸，你只需要更新模型并重新运行 `db.create_all()` 。然而，如果你在引擎表里记录了六个月的数据，你肯定不会想要从头开始。这时候就需要数据库迁移工具了。

Alembic是专用于SQLAlchemy的数据库迁移工具。它允许你保持你的数据库模式的版本历史，这样你就可以升级到一个新的模式，或者降级到旧的模式。

Alembic有一个可拓展的新手教程，所以我只会大概地说一下并指出一些需要注意的事项。

通过一个初始化的 `alembic init` 命令，你将创建一个alembic"迁移环境"。在你的版本库的根目录下执行这个命令，你将得到一个叫 `alembic` 的新文件夹。你的版本库将看上去就像Alembic教程中的这个例子一样：

```
myapp/
    alembic.ini
    alembic/
        env.py
        README
        script.py.mako
        versions/
            3512b954651e_add_account.py
            2b1ae634e5cd_add_order_id.py
            3adcc9a56557_rename_username_fiel
    myapp/
        __init__.py
        views.py
        models.py
          templates/
    run.py
    config.py
    requirements.txt
```

*alembic/*文件夹中包括了在版本间迁移数据的脚本。同时会有一个包括配置信息的*alembic.ini*文件。

> 注意 把*alembic.ini*添加到*.gitignore*中！在那里会有你的数据库凭证，所以你不应该把它留在版本控制中。
>
> 不过你可以把*alembic/*放进版本控制。它不会包含敏感信息（而且不能从你的源代码中重新生成），并且在版本控制中保存多个副本可以避免你的电脑发生不测。

当数据库模式需要发生变化时，我们需要做一系列事情。首先，运行 `alembic revision` 来生成迁移脚本。在*myapp/alembic/versions/*打开新生成的Python文件并使用Alembic的 `op` 对象完成 `upgrade` 和 `downgrade` 函数。

一旦我们的迁移脚本已经准备就绪，我们只需运行 `alembic upgrade head` 来迁移我们的数据到最新版本。

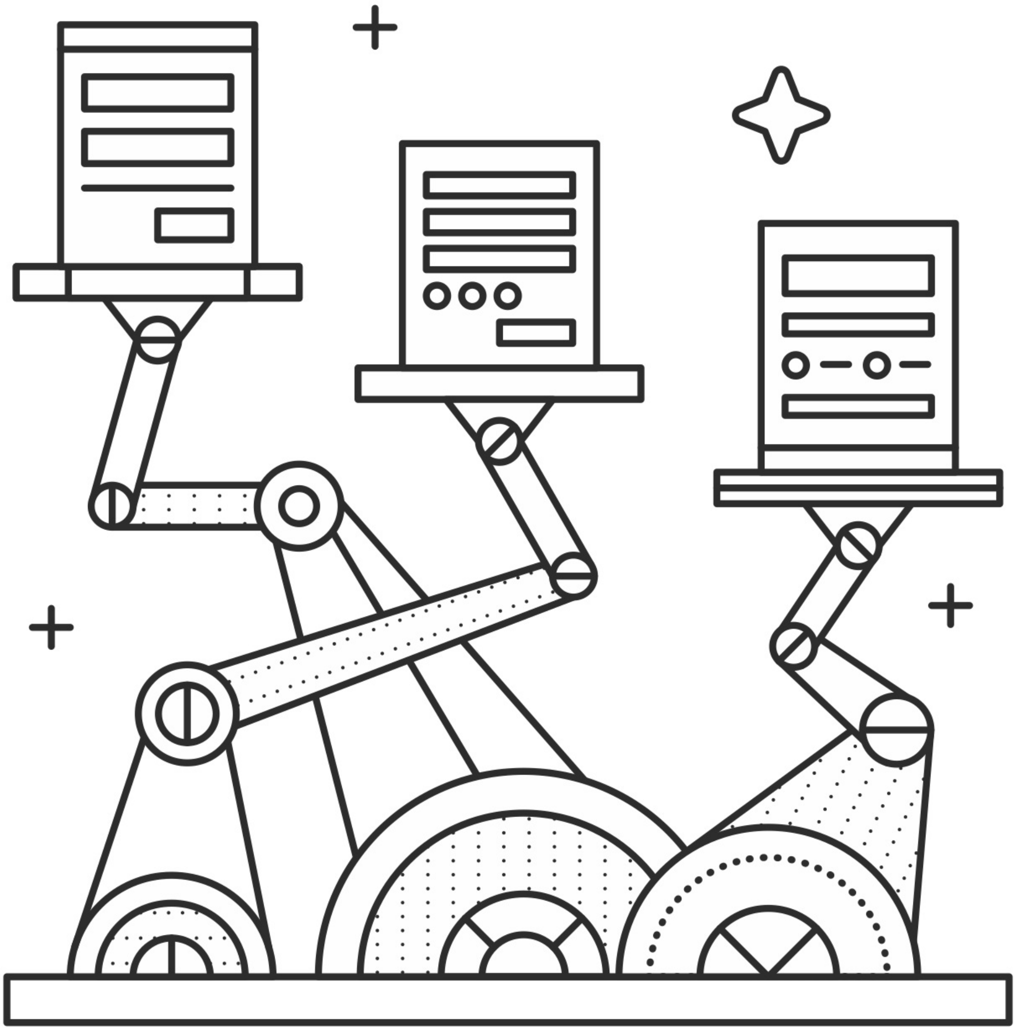
> 参见 想知道更多关于配置Alembic，创建你的迁移脚本和运行你的迁移，请看Alembic教程：http://alembic.readthedocs.org/en/latest/tutorial.html
>
> 注意 不要忘记设定数据的备份计划。备份计划的话题已经超出本书的范围，但你应该总是要有一个安全和健壮的方式备份你的数据库。
>
> 注意 Flask在NoSQL上的支持较少，但只要有你选择的数据库引擎有对应的Python库，你就能够用上它。这里有一些Flask插件，可以给Flask提供NoSQL引擎的支持。http://flask.pocoo.org/extensions/

# 总结

- 使用SQLAchemy来搭配关系型数据库。
- 使用Flask-SQLAlchemy来包装SQLAlchemy。
- Alembic会在数据库模式改变时帮助你管理数

据迁移。

- 你可以用NoSQL搭配Flask，但具体做法取决于具体引擎。
- 记得备份你的数据！



# 处理表单

表单是允许用户跟你的web应用交互的基本元素。Flask自己不会帮你处理表单，但Flask-WTF插件允许用户在Flask应用中使用脍炙人口的WTForms

包。这个包使得定义表单和处理表单功能变得轻松。

# Flask-WTF

你首要做的事（当然是在安装Flask-WTF之后），就是在 `myapp.forms` 包下定义一个表单类（form）。

*myapp/forms.py*

```
from flask_wtf import Form
from wtforms import StringField, PasswordFiel
from wtforms.validators import DataRequired, I

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[I
    password = PasswordField('Password', vali
```

> 注意 直到0.9版，Flask-WTF为WTForms的fields和validators提供自己的包装。你可能见过许多代码直接从 `flask_wtforms` 而不是 `wtforms` 中直接导入 `TextField` ， `PasswordField` 等等。 而从0.9版之后，我们得直接从 `wtforms` 中导入它们。

这个表单将用于用户注册表单。我们可以称之为 `SignInForm()` ，但是通过保持抽象，我们可以在别的地方重用它，比如作为登录表单。如果我们针对特定功能定义表单，最终就会得到许多相似却无法重用的表单。基于表单中包含的域 - 那些使得表单与众不同的元素 - 进行命名，显然会清晰很多。当然，有时候你会有复杂的，只在一个地方用到的表单，你再给它起个独一无二的名字也不迟。

这个表单可以帮我们做一些事情。它可以保护我们的应用免遭CSRF伤害，验证用户输入，为我们定义的域渲染适当的标记。

## CSRF保护和验证

CSRF全称是cross site request forgery，跨站请求伪造。CSRF通过第三方伪造表单数据，post到应用服务器上。受害服务器以为这些数据来自于它自己的网站，于是大意地中招了。

举个例子，假设你的邮件服务商允许你通过提交一个表单来注销账户。这个表单发送一个POST请求到他们服务器的 `account_delete` 页面，并且用户已经登录，就可以注销账户。你可以在你自己的网站中创建一个会发送到同一个 `account_delete` 页面的表单。现在，假如有个倒霉蛋点击了你的表单的'submit'（或者在他们加载你的页面的时候通过Javascript做到这一点），同时他们又登录了邮件账号，那么他们的账户就会被注销。除非你的邮件服务商知道不能假定提交过来的请求都是来自于自己的页面。

所以我们怎样判断一个POST请求是否来自我们自己的表单呢？WTForms在渲染每个表单时生成一个独一无二的token，使得这一切变得可能。那个token将在POST请求中随表单数据一起传递，并且会在表单被接受之前进行验证。关键在于token

的值取决于储存在用户的会话（cookies）中的一个值，而且会在一定时间之后过时（默认30分钟）。这样只有登录了页面的人（或至少是在那个设备之后的人）才能提交一个有效的表单，而且仅仅是在登录页面30分钟之内才能这么做。

> 参见
>
> - 这里是关于WTForms是怎么生成token的文档: http://wtforms.simplecodes.com/docs/1.0.1/ext.html#module-wtforms.ext.csrf.session
> - 这里有关于CSRF更多的信息: https://www.owasp.org/index.php/CSRF

为了开始使用Flask-WTF做CSRF防护，我们得先给我们的登录页定义一个视图。

myapp/views.py

```
from flask import render_template, redirect, u

from . import app
from .forms import EmailPasswordForm

@app.route('/login', methods=["GET", "POST"])
def login():
    form = EmailPasswordForm()
    if form.validate_on_submit():

        # Check the password and log the user
        # [...]

        return redirect(url_for('index'))
    return render_template('login.html', form=
```

我们从 `forms` 包中导入form对象，并于视图内实例化。然后运行 `form.validate_on_submit()` 。如果表单已经submit了（比如通过HTTP方法PUT或POST），这个函数返回 `True` 并且用定义在*forms.py*中的验证函数来验证表单。

> 参见 `validate_on_submit()` 的文档和源码在此：
>
> - http://flask-wtf.readthedocs.org/en/latest/api.html#flask_wtf.Form.validate_on_submit
> - https://github.com/ajford/flask-wtf/blob/v0.8.4/flask_wtf/form.py#L120

如果一个表单已经提交并且通过验证，我们可以开始处理登录逻辑的部分了。如果它还没有提交（比如，它只是一个GET请求），我们需要传递这个表单对象给模板来进行渲染。下面展示如何在模板中使用CSRF防护。

myapp/templates/login.html

```
{% extends "layout.html" %}
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <form action="{{ url_for('login') }}"
            <input type="text" name="email" /:
            <input type="password" name="passw
            {{ form.csrf_token }}
        </form>
    </body>
</html>
```

`{{ form.csrf_token }}` 将渲染一个隐藏的包括防范CSRF的特殊token的域，而WTForms会在验证表单时查找这个域。我们不用操心添加的任何特殊的验证token正确性的逻辑。万岁！

## 使用CSRFtoken来保护AJAX调用

Flask-WTF的CSRF token不仅限于保护表单提交。如果你的应用需要接受其他可能被伪造的请求（特别是AJAX调用），你也可以给它们添加

CSRF保护！想了解更多信息，请查看Flask-WTF的文档：https://flask-wtf.readthedocs.org/en/latest/csrf.html#ajax

# 自定义验证函数

除了WTForms提供的内置表单验证函数（比如`Required()`，`Email()`等等），你可以创建自己的验证函数。通过创建一个可用于检查数据库并确保用户提供的值未曾存在的`Unique()`验证函数，我将展示这一点。这个函数可以确保一个用户名或邮件地址未被使用。如果没有WTForms，我们不得不在视图中完成这些检查，但现在我们可以抽象出来作为form类的一部分。

*myapp/forms.py*

```
from flask_wtf import Form
from wtforms import StringField, PasswordFiel
from wtforms.validators import DataRequired, I

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[I
    password = PasswordField('Password', vali
```

现在我们想要添加一个验证函数来确认提供的邮件地址未曾出现在数据库中。我们将把验证函数放在一个新的 `util` 模块里，即 `util.validators` 。

*myapp/util/validators.py*

```
from wtforms.validators import ValidationErro

class Unique(object):
    def __init__(self, model, field, message=
        self.model = model
        self.field = field

    def __call__(self, form, field):
        check = self.model.query.filter(self.
        if check:
            raise ValidationError(self.messag
```

这个验证函数假定你是用SQLAlchemy来定义你的模型。WTForms要求验证函数返回可调用的(callable)类型（比如一个可调用的类）。

在*__init__.py*中，我们可以指定哪些参数应该传递给验证函数。在这个例子中我们需要检查相关的模型（比如`User`模型）和域。当验证函数被调用时，如果表单提交的值跟定义的模型的某个实例重复了，它会抛出一个`ValidationError`。我们也提供一个带默认值的信息参数，作为`ValidationError`的一部分。

现在我们给`EmailPasswordForm`添加`Unique`验证器。

*myapp/forms.py*

```
from flask_wtf import Form
from wtforms import StringField, PasswordFiel
from wtforms.validators import DataRequired, I

from .util.validators import Unique
from .models import User

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[I
    password = PasswordField('Password', vali
```

> 注意 你的验证函数不一定需要是可调用的类。它也可以是一个返回可调用对象的工厂类或者可调用对象。看这里的一些例子：[http://wtforms.simplecodes.com/docs/0.6.2/validators.html#custom-validators](http://wtforms.simplecodes.com/docs/0.6.2/validators.html#custom-validators)

## 渲染表单

WTForms也可以帮助我们给我们只需要表单渲染HTML。WTForms实现的 `Field` 类能根据域的形式渲染对应的HTML，所以我们只需要在模板中调

用它们。就像是渲染 `csrf_token` 域一样。下面是当我们使用WTForms来渲染我们的其他域时，login模板大概的样子。

myapp/templates/login.html

```
{% extends "layout.html" %}
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <form action="" method="POST">
            {{ form.email }}
            {{ form.password }}
            {{ form.csrf_token }}
        </form>
    </body>
</html>
```

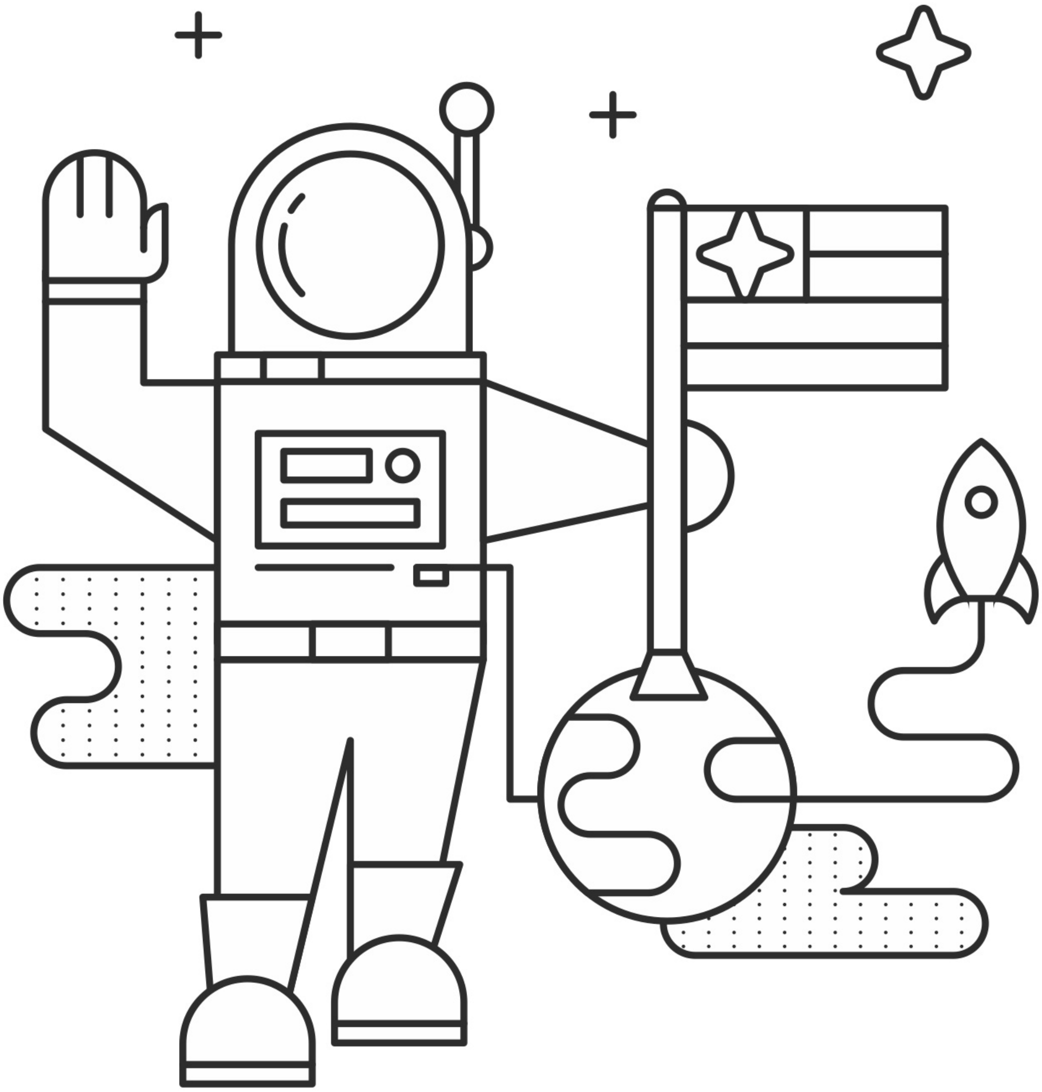
通过传递域的性质(properties)作为调用域的参数，我们可以自定义域的渲染形式。下面我们添加一个 `placeholder=` 性质给email域：

```
<form action="" method="POST">
    {{ form.email.label }}: {{ form.email(plac
    {{ form.password.label }}: {{ form.passwor
    {{ form.csrf_token }}
</form>
```

> 注意 如果我们想要传递HTML属性“class”， 我们得使用 `class_=''` ，因为“class”是Python的保留关键字。
>
> 参见 这个文档列出了所有可用的域性质：http://wtforms.simplecodes.com/docs/1.0.4/fields.html#wtforms.fields.Field.name
>
> 注意 你大概注意到了我们不需要使用Jinja的 `|safe` 过滤器。这是因为WTForms自己会处理掉HTML转义的问题。在这里了解更多信息： http://pythonhosted.org/Flask-WTF/#using-the-safe-filter

# 总结

- 表单可能会是安全上的阿喀琉斯之踵。
- WTForms（以及Flask-WTF）使得定义，保护和渲染你的表单更加轻松。
- 使用Flask-WTF提供的CSRF防范来保护你的表单。
- 你也可以使用Flask-WTF来防止AJAX调用遭到CSRF攻击。
- 定义自定义的表单验证函数，避免在视图函数中写入验证逻辑。
- 使用WTForms的域渲染功能来渲染你的表单的HTML，这样每次修改表单的定义时，你不需要更新模板。



# 用户管理的规范

用户管理是现代Web应用都需要做的事情之一。一个仅有基本的账户功能的应用也需要处理一大堆诸如注册，邮件确认，安全地存储密码，重置

密码，用户验证以及更多。考虑到许多安全问题都出现在管理用户时，在这个领域最好遵循普遍的规范。

> 注意 在本章中我会假定你已经在用SQLAlchemy模型和WTForms来处理你的表单输入。如果你不使用它们，你需要修改这些规范来适应你喜欢的方法。

# 邮件确认

当一个新用户给你他们的邮件地址，你通常需要确认该地址是否是正确的。一旦你完成了验证，你就可以安心地发送密码重置链接和其他敏感信息给该邮箱，不用担心位于接收端的会是谁。

邮件确认的一个通常的规范是发送一个当前独一无二的URL密码重置链接，来确认用户的电子邮件地址。举个例子，john@gmail.com注册了你的应用。你的应用把他登记在数据库中，设置 `email_confirmed` 列为 `False` 并发送一封带特

定URL的邮件给john@gmail.com。这个URL通常包括一个独一无二的token，比如 http://myapp.com/accounts/confirm/kj3kjhj3hj3。当John收到那封邮件时，他点击链接。你的应用看到了token，知道是哪封邮件并设置John的 `email_confirmed` 列为 `True` 。

那我们怎么知道给定的token对应的是哪封邮件？一个方法是在创建token时把它存储到数据库中，在我们收到一个确认请求时检索数据库来找到那个token。这需要做很多事情，而幸运的是，我们不必这么做。

我们将邮件地址编码进token。它还包括一个时间戳，表示这个token的有效期。为了做到这一点，我们要使用 `itsdangerous` 包。这个包提供了在无法信赖的环境中发送敏感信息的工具。（比如发送邮件确认token给未验证的邮件地址）。在这个例子里，我们将使用 `URLSafeTimedSerializer` 。

*myapp/util/security.py*

```
from itsdangerous import URLSafeTimedSerialize

from .. import app

ts = URLSafeTimedSerializer(app.config["SECRET
```

现在当用户给我们邮件地址时，我们可以使用这个序列器来生成验证token。通过这种方式，我们来实现一个简单的账户注册流程。

*myapp/views.py*

```
from flask import redirect, render_template, u

from . import app, db
from .forms import EmailPasswordForm
from .util import ts, send_email

@app.route('/accounts/create', methods=["GET",
def create_account():
    form = EmailPasswordForm()
    if form.validate_on_submit():
        user = User(
            email = form.email.data,
            password = form.password.data
        )
        db.session.add(user)
        db.session.commit()

        # Now we'll send the email confirmatio
```

```
            subject = "Confirm your email"

            token = ts.dumps(self.email, salt='ema

            confirm_url = url_for(
                'confirm_email',
                token=token,
                _external=True)

            html = render_template(
                'email/activate.html',
                confirm_url=confirm_url)

            # 假设在myapp/util.py中定义了send_mail
            send_email(user.email, subject, html)

            return redirect(url_for("index"))

        return render_template("accounts/create.ht
```

这段视图实现了创建用户并发送邮件到给定的邮件地址。你可能注意到了，我们使用一个模板来给电子邮件生成HTML。我们来看看这个电子邮件模板的例子。

*myapp/templates/email/activate.html*

```
你的账户已经成功创建<br>
请点击打开以下链接来激活你的邮箱：

<p>
<a href="{{ confirm_url }}">{{ confirm_url }}
</p>

<p>
--<br>
如果对本邮件有疑问或者有话想说，发邮件给hello@myapp.
</p>
```

OK，所以现在我们只需要实现一个处理那个邮件中的验证链接的视图。

*myapp/views.py*

```
@app.route('/confirm/<token>')
def confirm_email(token):
    try:
        email = ts.loads(token, salt="email-co
    except:
        abort(404)

    user = User.query.filter_by(email=email).f

    user.email_confirmed = True

    db.session.add(user)
    db.session.commit()

    return redirect(url_for('signin'))
```

这个视图只是一个简单的表单视图。我们仅仅在开头添加了 `try ... except` 来检查这个token是否有效。这个token包括一个时间戳，所以我们可以调用 `ts.loads()` ，如果它比 `max_age` 还大，就抛出一个异常。在这个例子，我们设置 `max_age` 为86400秒，也即24小时。

> 注意 你可以用差不多的方法实现一个邮件重置的功能。仅需要发送带旧邮件地址和新地址的token的验证链接到新的邮件地址。如果token是有效的，用新的地址更新旧地址。

# 存储密码

用户管理的第一条军规是在存储它们之前使用Bcrypt算法（或者scrypt，不过这里我们将使用Bcrypt）hash密码。你绝不可明文存储密码。这会是严重的安全问题并且它损害了你的用户。所有的繁重工作都已经有第三方的包来完成，所以没有任何不遵循这个最佳实践的理由。

> 参见 OWASP是业界最值得信赖的关于Web应用安全的信息来源之一。看一下他们推荐的一些安全编程规范：
> https://www.owasp.org/index.php/Secure_Coding_Cheat_Sheet#Password_Storage

我们将继续前进，使用Flask-Bcrypt插件来实现应用中的bcrypt包。这个插件只是基于 `py-bcypt` 包的包装，但是它帮我们处理了一些琐碎的事（比如在比较hash结果之前检查字符串编码）。

myapp/__init__.py

```
from flask_bcrypt import Bcrypt

bcrypt = Bcrypt(app)
```

Bcrypt算法之所以深受欢迎，其中一个原因是它的“未来拓展性”。这意味着随着时间的迁移，当计算能力越来越廉价时，我们可以让它越来越难通过暴力算法来测试成百上千万密码组合来破解。我们用于hash密码的"rounds"越多，完成一次尝试所花费的时间就越长。如果在存储密码前，我们把它hash了20次，骇客也不得不hash他们的每次猜测20次。

记住如果我们hash密码20次，需要等到计算结束之后，我们的应用才会做出响应。这意味着，在选择计算的次数时，我们要取得安全性和可用性的一个平衡点。在给定时间内你能计算的次数取决于你拥有的计算资源，所以最好测试不同的数字，找到能在0.25到0.5秒间完成一个密码的hash的值。至少，先从12次（12 rounds）开始尝试吧。

要想测试hash一个密码的时间，你可以 `time` 一个简单的，用于hash一个密码的Python脚本看看。

*benchmark.py*

```
from flask_bcrypt import generate_password_has

# 改变round的次数（第二个参数），直到运行时间在0.25到
generate_password_hash('password1', 12)
```

现在我们可以用 `time` 命令测几次看看。

```
$ time python test.py

real    0m0.496s
user    0m0.464s
sys     0m0.024s
```

我曾在一个小服务器上做过快速的基准测试，发现12 rounds正好能花费恰当的时间，所以我在这个例子中这么配置。

config.py

```
BCRYPT_LOG_ROUNDS = 12
```

既然Flask-Bcrypt已经配置完毕了，是时候开始hash密码。我们本可以在接受注册表单的视图函数中手工完成，但是将来在密码重置和密码修改视图中，同样的代码还得一再重复。所以，我们需要抽象hash的过程，这样即使我们忘记了，我们的应用也会悄悄完成它。秘诀在于我们写了个**setter**，这样当设置 `user.password = 'password1'` 时，密码在存储之前就会被用Bcrypt自动hash了。

myapp/models.py

```
from sqlalchemy.ext.hybrid import hybrid_prope

from . import bcrypt, db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=Tru
    username = db.Column(db.String(64), unique
    _password = db.Column(db.String(128))

    @hybrid_property
    def password(self):
        return self._password

    @password.setter
    def _set_password(self, plaintext):
        self._password = bcrypt.generate_passw
```

我们使用SQLAlchemy的hybrid（混合）拓展来定义一个同时供众多函数调用的接口属性。当赋值给 `user.password` 属性时，我们的setter会被自动调用。而在setter内，我们会hash纯文本密码并存储在用户表里的 `_password` 列里。既然我们定义 `user.password` 为混合属性，那么就可以通过这个属性来获取 `_password` 的值。

现在我们用这个模型来实现注册视图。

myapp/views.py

```
from . import app, db
from .forms import EmailPasswordForm
from .models import User

@app.route('/signup', methods=["GET", "POST"]
def signup():
    form = EmailPasswordForm()
    if form.validate_on_submit():
        user = User(username=form.username.da
        db.session.add(user)
        db.session.commit()
        return redirect(url_for('index'))

    return render_template('signup.html', for
```

# 验证

既然把用户加入到数据库中了，就可以实现验证功能了。我们想要让用户通过表单提交他们的用户名和密码（当然，有些时候是邮箱和密码），然后验证他们提供的密码是否正确。如果一切安好，我们将通过设置浏览器的cookie来标记他们是已验证的用户。下一次他们再提交请求时，通过查看cookie，我们就知道他们已经登录过了。

先从用WTForms定义一个`UsernamePassword`开始吧。

myapp/forms.py

```
from flask_wtf import Form
from wtforms import StringField, PasswordFiel
from wtforms.validators import DataRequired

class UsernamePasswordForm(Form):
    username = StringField('Username', valida
    password = PasswordField('Password', vali
```

接下来我们将往我们的用户模型添加一个方法，拿一个字符串跟已存储的hash过的用户密码作比较。

myapp/models.py

```
from . import db

class User(db.Model):

    # [...] columns and properties

    def is_correct_password(self, plaintext)
        if bcrypt.check_password_hash(self._pa
            return True

        return False
```

# Flask-Login

我们下一个目标是定义一个使用我们的表单类的登录视图。如果用户输入正确的账号，我们将使用Flask-Login插件来验证它们。这个插件简化了处理用户会话和验证的操作。

我们只需做少量的配置就能让Flask-Login用起来了。

我们先在*__init__.py*定义Flask-Login的 `login_manager` 。

*myapp/__init__.py*

```
from flask_login import LoginManager

# 创建并配置应用
# [...]

from .models import User

login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view =  "signin"

@login_manager.user_loader
def load_user(userid):
    return User.query.filter(User.id == useri
```

我们在这里创建一个叫 `LoginManager` 的实例，用我们的 `app` 对象初始化它，定义登录视图并告诉它如何通过 `id` 获取用户类。这是使用Flask-Login的基本配置。

> 参见 你可以在这里找到自定义Flask-Login的更多信息：https://flask-login.readthedocs.org/en/latest/#customizing-the-login-process

现在我们来定义处理验证的 `signin` 视图。

*myapp/views.py*

```
from flask import redirect, url_for

from flask_login import login_user

from . import app
from .forms import UsernamePasswordForm()

@app.route('signin', methods=["GET", "POST"])
def signin():
    form = UsernamePasswordForm()

    if form.validate_on_submit():
        user = User.query.filter_by(username=
        if user.is_correct_password(form.pass
            login_user(user)

            return redirect(url_for('index'))
        else:
            return redirect(url_for('signin'))
    return render_template('signin.html', for
```

我们仅需要从Flask-Login import `login_user` 函数，检查用户的验证信息，并调用 `login_user(user)` 。你使用 `logout_user()` 登出当前用户。

*myapp/views.py*

```
from flask import redirect, url_for
from flask_login import logout_user

from . import app

@app.route('/signout')
def signout():
    logout_user()

    return redirect(url_for('index'))
```

# 忘记密码？

你总会需要实现一个“忘记密码？”功能来允许用户通过邮件重置自己的账号密码。这个地方可能会有潜在安全隐患，因为你不得不让一个未验证的用户接管一个账户。我们将会使用类似于邮件验证的方式来实现密码重置的功能。

我们将需要一个表单类来请求对给定账户的重置，还有一个表单来选择一个新的密码（前提是未验证用户访问了账户邮箱）。这里假设我们的用户模型有一个email和一个password，而password是我们之前设置过的混合属性。

> 注意 不要发送密码重置链接给未确认的邮箱！你要确保发送链接给正确的人。

我们将需要两个表单。一个用于请求一个重置链接，另一个用于在通过验证之后修改密码。

myapp/forms.py

```
from flask_wtf import Form

from wtforms import StringField, PasswordFiel
from wtforms.validators import DataRequired, I

class EmailForm(Form):
    email = StringField('Email', validators=[I

class PasswordForm(Form):
    password = PasswordField('Email', validat
```

假设我们的密码重置表单只需要密码这一栏。许多应用需要用户两次输入他们的新密码，确保没有打错。为了实现这个，我们仅需添加另一个 `PasswordField` ，并加一个WTForms验证函数 `EqualTo` 到主密码域。

> 参见 很多人站在用户体验的角度，对什么是设计注册表单的最佳方式有过许多有趣的讨论。我个人喜欢Stack Exchange用户 Roger Attrill说的一番话：“我们不应该一再要求用户输入密码 - 我们应该要求输入一次，然后确保‘忘记密码’能无缝且正确地运行。”
>
> - 你可以在User Experience Stack Exchange读到更多关于这个话题的内容:http://ux.stackexchange.com/questions/20953/why-should-we-ask-the-password-twice-during-registration/21141
> - 在Smashing Magazine的文章中，你可以读到一些简化注册和登录表单的酷想法：http://uxdesign.smashingmagazine.com/2011/05/05/innovative-techniques-to-simplify-signups-and-logins/

现在我们将开始迈出第一步，让用户可以请求发送一个密码重置链接给绑定的邮箱地址。

myapp/views.py

```
from flask import redirect, url_for, render_te

from . import app
from .forms import EmailForm
from .models import User
from .util import send_email, ts

@app.route('/reset', methods=["GET", "POST"])
def reset():
    form = EmailForm()
    if form.validate_on_submit()
        user = User.query.filter_by(email=forn

        subject = "Password reset requested"
        # Here we use the URLSafeTimedSerializ
        token = ts.dumps(user.email, salt='rec

        recover_url = url_for(
            'reset_with_token',
            token=token,
            _external=True)

        html = render_template(
            'email/recover.html',
            recover_url=recover_url)

        # Let's assume that send_email was def
        send_email(user.email, subject, html)

        return redirect(url_for('index'))
    return render_template('reset.html', form=
```

当表单接受到一个邮件地址时，我们取出对应的用户，生成一个重置token，再发送一个重置密码URL给用户。这个URL将引导用户前往验证token的视图，并让用户重置密码。

myapp/views.py

```
from flask import redirect, url_for, render_te

from . import app, db
from .forms import PasswordForm
from .models import User
from .util import ts

@app.route('/reset/<token>', methods=["GET", '
def reset_with_token(token):
    try:
        email = ts.loads(token, salt="recover
    except:
        abort(404)

    form = PasswordForm()

    if form.validate_on_submit():
        user = User.query.filter_by(email=ema

        user.password = form.password.data

        db.session.add(user)
        db.session.commit()

        return redirect(url_for('signin'))

    return render_template('reset_with_token.h
```

我们将使用验证用户邮箱时用的那个token验证方式。这个视图传递token回模板，然后模板会在表单中提交正确的URL。让我们看看这个模板到底

长啥样。

*myapp/templates/reset_with_token.html*

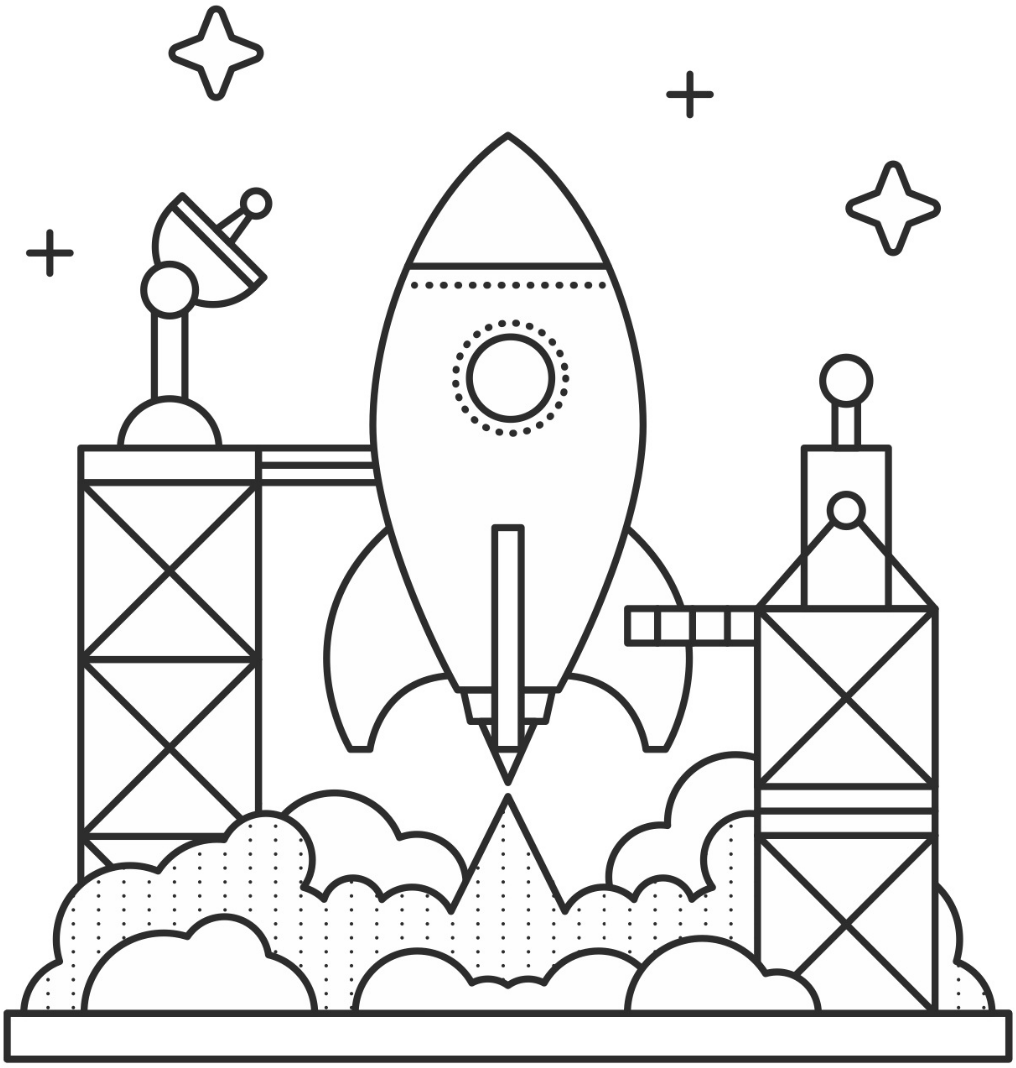
```
{% extends "layout.html" %}

{% block body %}
<form action="{{ url_for('reset_with_token', t
    {{ form.password.label }}: {{ form.passwor
    {{ form.csrf_token }}
    <input type="submit" value="Change my pass
</form>
{% endblock %}
```

# 总结

- 使用itsdangerous包来创建和验证送往邮箱的token。
- 你可以使用token来验证邮箱，无论是在用户注册账户，还是修改邮箱，或者忘记密码的时候。
- 使用Flask-Login插件来验证用户，这样能避免处理一堆会话管理的麻烦事。
- 总是设想会有恶意的用户试图从应用中挖掘

漏洞。



# 部署

最终，你终于可以向全世界展示你的应用了。是时候部署它了。这个过程总能让人感到受挫，因为有太多任务需要完成。同时在部署的过程中你

需要做出太多艰难的决定。我们会谈论一些关键的地方以及我们一些可能的选择。

# 托管主机

首先，你需要一个服务器。世上服务器提供商成千，但我只取三家。我不会谈论如何开始使用它们的服务的细节，因为这超出本书的范围。相反，我只会谈论它们作为Flask应用托管商上的优点。

## Amazon Web Services EC2（因为国情问题，让我们直接看下一个吧）

Amazon Web Services指的是一套相关的服务，提供商是……~~卓越~~亚马逊！今日，许多著名的初创公司选择使用它，所以你或许已经听过它的大名。AWS服务中我们最关心的是EC2，全称是Elastic Compute Cloud。EC2的最大的卖点是你

能够获得虚拟主机，或者说实例（这是AWS官方称呼），在仅仅几秒之内。如果你需要快速拓展你的应用，就只需启动多一点EC2实例给你的应用，并且用一个负载平衡器(load balancer)管理它们。（这时还可以试试AWS Elastic Load Balancer）

对于Flask而言，AWS就是一个常规的虚拟主机。付上一些费用，你可以用你喜欢的Linux发行版启动它，并安上你的Flask应用。之后你的服务器就起来了。不过它意味着你需要一些系统管理知识。

# Heroku

Heroku是一个应用托管网站，基于诸如EC2的AWS的服务。他们允许你获得EC2的便利，而无需系统管理经验。

对于Heroku，你通过 `git push` 来在它们的服务器上部署代码。这是非常便利的，如果你不想浪费时间ssh到服务器上，安装并配置软件，继续整个

常规的部署流程。这种便利是需要花钱购买的，尽管AWS和Heroku都提供了一定量的免费服务。

> 参见 Heroku有一个如何在它们的服务器上部署Flask应用的教程：https://devcenter.heroku.com/articles/getting-started-with-python
>
> 注意 管理你自己的数据库将会花上许多时间，而把它做好也需要一些经验。通过配置你自己的站点来学习数据库管理是好的，但有时候你会想要外包给专业团队来省下时间和精力。Heroku和AWS都提供有数据库管理服务。我个人还没试过，但听说它们不错。如果你想要保障数据安全以及备份，却又不想要自己动手，值得考虑一下它们。
>
> - Heroku Postgres: https://www.heroku.com/postgres
> - Amazon RDS: https://aws.amazon.com/rds/

# Digital Ocean

Digital Ocean是最近出现的EC2的竞争对手。一如EC2，Digital Ocean允许你快速地启动虚拟主机（在这里叫droplet）。所有的droplet都运行在SSD上，而在EC2，如果你用的是普通服务，你是享受不到这种待遇的。对我而言，最大的卖点是它提供的控制接口比AWS控制面板简单和容易多了。Digital Ocean是我个人的最爱，我建议你考虑下它。

在Digital Ocean，Flask应用部署方式就跟在EC2一样。你会得到一个全新的Linux发行版，然后需要安装你的全套软件。

# 部署工具

这一节将包括一些为了向别人提供服务，你需要安装在服务器上的软件。最基本的是一个前置服务器，用来反向代理请求给一个运行你的Flask应

用的应用容器。你通常也需要一个数据库，所以我们也会略微谈论下这方面的内容。

## 应用容器

在开发应用时，本地运行的那个服务器并不能处理真实的请求。当你真的需要向公众发布你的应用，你需要在应用容器，例如Gunicorn，上运行它。Gunicorn接待请求，并处理诸如线程的复杂事务。

要想使用Gunicorn，需要通过pip安装`gunicorn`到你的虚拟环境中。运行你的应用只需简单的命令。为了简明起见，让我们假设这就是我们的Flask应用：

*app.py*

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello World!"
```

哦，这真简明扼要。现在，使用Gunicorn来运行它吧，我们只需执行这个命令：

```
(ourapp)$ gunicorn rocket:app
2014-03-19 16:28:54 [62924] [INFO] Starting gu
2014-03-19 16:28:54 [62924] [INFO] Listening a
2014-03-19 16:28:54 [62924] [INFO] Using worke
2014-03-19 16:28:54 [62927] [INFO] Booting wor
```

你应该能在 http://127.0.0.1:8000 看到“Hello World!”。

为了在后台运行这个服务器（也即使它变成守护进程），我们可以传递 `-D` 选项给Gunicorn。这下它会持续运行，即使你关闭了当前的终端会话。

如果我们这么做了，当我们想要关闭服务器时就会困惑于到底应该关闭哪个进程。我们可以让Gunicorn把进程ID储存到文件中，这样如果想要停止或者重启服务器时，我们可以不用在一大串运行中的进程中搜索它。我们使用 `-p <file>` 选项来这么做。现在，我们的Gunicorn部署命令是这样：

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -
(ourapp)$ cat rocket.pid
63101
```

要想重新启动或者关闭服务器，我们可以运行对应的命令：

```
(ourapp)$ kill -HUP `cat rocket.pid` # 发送一个
(ourapp)$ kill `cat rocket.pid`
```

默认下Gunicorn会运行在8000端口。如果这已经被另外的应用占用了，你可以通过添加 `-b` 选项来指定端口。

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -l
```

## 将Gunicorn摆上前台

> 注意 Gunicorn应该隐藏于反向代理之后。如果你直接让它监听来自外网的请求，它很容易成为拒绝服务攻击的目标。它不应该接受这样的考验。只有在debug的情况下你才能把Gunicorn摆上前台，而且完工之后，切记把它重新隐藏到幕后。}

如果你像前面说的那样在服务器上运行Gunicorn，将不能从本地系统中访问到它。这是因为默认情况下Gunicorn绑定在127.0.0.1。这意味着它仅仅监听来自服务器自身的连接。所以通常使用一个反向代理来作为外网和Gunicorn服务器的中介。不过，假如为了debug，你需要直接从外网发送请求给Gunicorn，可以告诉Gunicorn绑定0.0.0.0。这样它就会监听所有请求。

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -l
```

> 注意
>
> - 从文档中可以读到更多关于运行和部署Gunicorn的信息：http://docs.gunicorn.org/en/latest/
> - Fabric是一个可以允许你不通过SSH连接到每个服务器上就可以执行部署和管理命令的工具：http://docs.fabfile.org/en/latest

## Nginx反向代理

反向代理处理公共的HTTP请求，发送给Gunicorn并将响应带回给发送请求的客户端。Nginx是一个优秀的客户端，更何况Gunicorn强烈建议我们使用它。

要想配置Nginx作为运行在127.0.0.1:8000的Gunicorn的反向代理，我们可以在*/etc/nginx/sites-available*下给应用创建一个文

件。不如称之为*exploreflask.com*吧。

*/etc/nginx/sites-available/exploreflask.com*

```
# Redirect www.exploreflask.com to explorefla
server {
        server_name www.exploreflask.com;
        rewrite ^ http://exploreflask.com/ pe
}

# Handle requests to exploreflask.com on port
server {
        listen 80;
        server_name exploreflask.com;

        # Handle all locations
        location / {
                # Pass the request to Gunicor
                proxy_pass http://127.0.0.1:8

                # Set some HTTP headers so th
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $r
                proxy_set_header X-Forwarded-
        }
}
```

现在在*/etc/nginx/sites-enabled*下创建该文件的符号链接，接着重启Nginx。

```
$ sudo ln -s \
/etc/nginx/sites-available/exploreflask.com \
/etc/nginx/sites-enabled/exploreflask.com
```

你现在应该可以发送请求给Nginx然后收到来自应用的响应。

> 参见 Gunicorn文档中关于配置Nginx的部分会给你更多启动Nginx的信息: http://docs.gunicorn.org/en/latest/deploy.html#nginx-configuration

## ProxyFix

有时，你会遇到Flask不能恰当处理转发的请求的情况。这也许是因为在Nginx中设置的某些HTTP报文头部造成的。我们可以使用Werkzeug的ProxyFix来fix转发请求。

*app.py*

```
from flask import Flask

# Import the fixer
from werkzeug.contrib.fixers import ProxyFix

app = Flask(__name__)

# Use the fixer
app.wsgi_app = ProxyFix(app.wsgi_app)

@app.route('/')
def index():
    return "Hello World!"
```

> 参见 在Werkzeug文档中可以读到更多关于ProxyFix的信息: http://werkzeug.pocoo.org/docs/contrib/fixers/#werkzeug.contrib.fixers.ProxyFix

# 总结

- 你可以把Flask应用托管到AWS EC2，Heroku和Digital Ocean。（译者注：建议托管到国内的云平台上）
- Flask应用的基本部署依赖包括一个应用容器

（比如Gunicorn）和一个反向代理（比如Nginx）。

- Gunicorn应该退居Nginx幕后并监听127.0.0.1（内部请求）而非0.0.0.0（外部请求）
- 使用Werkzeug的ProxyFix来处理Flask应用遇到的特定的转发报文头部。